Denotational semantics and proof methods for a lazy functional language

Hendrik Wietze de Haan

Supervisor: W.H. Hesselink

RuG

Computer Science

Masters thesis

Denotational semantics and proof methods for a lazy functional language

Hendrik Wietze de Haan

Supervisor: W.H. Hesselink

Rijksuniversiteit Groningen
Computer Science
Postbus 800
9700 AV Groningen

September 2001

Contents

Chapter 1

Introduction

Formal description of a language gives insight into the language itself. The formal description may point out inconsistencies in the language, result in new program constructions or/and deliver useful proof methods. The formal description can be divided into the structure or syntax of programs in the language and semantics or meaning of programs.

The semantics of a language can be described in many ways. An operational semantics describes the meaning or behavior of a program by means of (operational) arguments based on the execution of the program. A denotational semantics models the program as a mathematical object in a domain of possible meanings. This object can then be understood by mathematical tools and rules. There are more semantic models and the relations between these models have been studied by many people. A model is fully abstract when all the objects in the model correspond to programs in the language. Proving such is quite intricate. A good overview of the current status of the research into denotational semantics and the problem of full abstractness can be found in [Jun96].

In this paper we shall concentrate on the denotational semantics of a functional language. From the mathematics of the semantics we will gain insight into some proof methods which are connected to a denotational semantics.

1.1 Outline of the paper

Chapter 2 is a quick introduction to domain theory. The necessary mathematical constructions for specifying a denotational semantics is presented. The chapter is based on chapters 8 and 12 of [Win92], and the omitted proofs can be found there.

Chapter 3 presents the language Lapa which was the starting point for our paper. In this chapter we define the characteristics of Lapa and try to formalize a denotational semantics. The chapter concludes with the deficiencies of Lapa and their solutions.

Chapter 4 presents the language Lager which is a possible language one might acquire if the deficiencies of Lapa are solved. The formal definition of Lager including an operational and a denotational semantics are presented here.

Chapter 5 reviews some proof methods. The soundness of the proof principle for properties of infinite, partial and finite lists found in chapter 9 of [Bir98] is investigated. The approximation lemma which can also be found in chapter 9 of [Bir98] is also looked into. The chapter concludes with a brief remark on other proof methods. It is largely a brief summary of the article [GH99].

Chapter 6 presents some conclusions and points out further work.

Appendix A gives an implementation of Lager in Haskell based on monads. Roughly the same implementation in Gofer without monads can be found in Appendix B.

Chapter 2

Domain Theory

In this chapter we shall introduce the necessary theory to define a denotational semantics for Lager. This will include the basic definitions of e.g. (complete) partial orders, continuous functions and fixpoints, but also information systems and constructions on those information systems. Only the necessary constructions are described. The more general versions of these constructions may be found in the chapters 8 and 12 of [Win92].

2.1 Notations

Definition. Let A and B be two sets. The disjunct union $A \uplus B$ of A and B is defined as

$$A \uplus B = (\{1\} \times A) \cup (\{2\} \times B)$$

Definition. Let x be a variable, e be an expression and let t be an entity that is to be substituted for each free occurrence of x in e. We denote the subtitution of t for x in e by $e[t/x]$. Simultaneous substitution can also be expressed. Let $x_1, \ldots, x_n$ be n distinct variables for a natural number n. Let $t_1, \ldots, t_n$ be n entities and let e be an expression. With $e[t_1/x_1, \ldots, t_n/x_n]$ we denote the simultaneous substitution of t_1 for each free occurence of x_1, ..., t_n for each free occurence of x_n in e.

Remark. We will not only apply substitutions to syntactical expressions, but also to mathematical functions. When applied to a function f, a substitution $f[v/x]$ updates the function at the place of the argument x with the new value v, i.e.

$$(f[v/x])(y) = \begin{cases} f(y) & \text{if } y \not\equiv x \\ v & \text{if } y \equiv x \end{cases}$$

We employ rules to present operational semantics, typing rules and more.

Definition. A *rule* has some premisses and a conclusion. A rule without premisses is called an axiom and holds trivially. Usually a rule is written in the following form

$$\frac{\text{premisses}}{\text{conclusion}}$$

Rules can be used to construct so-called proof trees. To prove the conclusion one needs to prove the premisses, which may involve repeated applications of rules.

2.2 Basic definitions

Definition. A *partial ordered set (poset)* is tuple $(P, \sqsubseteq)$, where P is a set and $\sqsubseteq$ a binary relation on P which is

1. reflexive: $(\forall\, p \in P :: p \sqsubseteq p)$
2. antisymmetric: $(\forall\, p, q \in P : p \sqsubseteq q \wedge q \sqsubseteq p : p = q)$
3. transitive: $(\forall\, p, q, r \in P : p \sqsubseteq q \wedge q \sqsubseteq r : p \sqsubseteq r)$

Definition. Let $(P, \sqsubseteq)$ be a poset and let $X \subseteq P$.

- An element $x \in X$ is a *smallest* element of X iff $(\forall\, y \in X :: x \sqsubseteq y)$.
- An element $p \in P$ is an *upperbound* of X iff $(\forall\, q \in X :: q \sqsubseteq p)$.
- An element $p \in P$ is a *least upperbound (lub)* or supremum of X iff
  1. $(\forall\, q \in P :: q \sqsubseteq p)$
  2. $(\forall\, r \in P : (\forall\, q \in X :: q \sqsubseteq r) : p \sqsubseteq r)$

  If X has a least upperbound, we denote this least upper bound by $\bigsqcup X$

Definition. Let $(D, \sqsubseteq)$ be a poset. An *ω-chain* of the poset is an ascending chain

$$d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$$

of elements of the poset.

In the theory of partial orders it is customary to identify the set of natural numbers $\mathbb{N}$ as ω, hence the name *omega*-chain or ω-chain.

Definition. A *complete partial order (cpo)* is a poset $(D, \sqsubseteq)$ in which every ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$ of the poset has a least upperbound (or *limit*) in D, i.e. $\bigsqcup\{d_n \mid n \in \omega\} \in D$.

A *cpo with bottom* is a cpo $(D, \sqsubseteq_D)$ with a smallest element, which is usually $\bot_D$.

Definition. Any set ordered by the equality relation forms a cpo. Such cpos are called *discrete* cpos. A *flat* cpo is a lifted discrete cpo, i.e. a discrete cpo to which a bottom element has been added.

Remark. The set $\mathbb{Z}$ of integers ordered by the equality relation forms a cpo which we shall denote by $\mathbf{Z}$. The flat cpo $\mathbf{Z}_\bot$ is obtained from $\mathbf{Z}$ by means of lifting (lifting shall be introduced in section 2.3.1).

Definition. Let $f : D \to E$ be a function from cpo $(D, \sqsubseteq_D)$ to cpo $(E, \sqsubseteq_E)$.

- f is *monotone* iff $(\forall\, d_0, d_1 \in D : d_0 \sqsubseteq_D d_1 : f(d_0) \sqsubseteq_E f(d_1))$
- f is *continuous* iff
  1. f is monotone
  2. for each ω-chain $d_0 \sqsubseteq_D d_1 \sqsubseteq_D \cdots \sqsubseteq_D d_n \sqsubseteq_D \cdots$ in D we have

$$\bigsqcup_{n \in \omega} f(d_n) = f(\bigsqcup_{n \in \omega} d_n)$$

We generally denote a cpo by its set and leave out the ordering, i.e. we write D instead of $(D, \sqsubseteq_D)$. We leave out the label of the ordering when it is clear to which cpo the ordering belongs.

Theorem. Let $f : D \to D$ be a continuous function on a cpo with bottom $(D, \sqsubseteq)$. Define

$$\mathrm{fix}(f) = \bigsqcup_{n \in \omega} f^n(\bot)$$

Then

1. $f(\mathrm{fix}(f)) = \mathrm{fix}(f)$
2. $f(d) \sqsubseteq d \Rightarrow \mathrm{fix}(f) \sqsubseteq d$

In other words, $\mathrm{fix}(f)$ is the least (pre-)fixpoint of f.

Proof. This is a well known fact and can be found in e.g. chapter 8 of [Win92], or in any good text on domain theory. □

Definition. Let D and E be cpos. D and E are *isomorphic* if there is an *isomorphism* between both cpos. A continuous function $f : D \to E$ is an *isomorphism* iff f is a bijection that satifies

$$(\forall\, x, y \in D :: x \sqsubseteq_D y \;\Leftrightarrow\; f(x) \sqsubseteq_E f(y))$$

When cpos D and E are ismorphic we write $D \cong E$.

2.3 Constructions on cpos

From basic cpos, like $\mathbf{Z}$, new cpos can be constructed by combining them in various ways, like products, sums and functions from cpos to cpos. These basic cpo-operations are used to build the cpos needed for the denotational semantics of Lager.

2.3.1 Lifting

Lifting a cpo adds a new bottom element below a copy of the original elements of the cpo. Besides the new element we need a function that makes a 'copy' of the old elements.

Definition. Let D be a cpo. Assume that $\perp$ is a new element. Assume that $\lfloor - \rfloor$ is a function that satisfies

1. $(\forall\, d_0, d_1 \in D :: \lfloor d_0 \rfloor = \lfloor d_1 \rfloor \Rightarrow d_0 = d_1)$
2. $(\forall\, d \in D :: \perp \neq \lfloor d \rfloor)$

We define the *lifted* cpo $D_\perp$ as the cpo with underlying set

$$\{\lfloor d \rfloor \mid d \in D\} \cup \{\perp\}$$

and partial order

$$d'_0 \sqsubseteq_{D_\perp} d'_1 \;\equiv\; (d'_0 = \perp) \;\vee\; (\exists\, d_0, d_1 \in D : d_0 = \lfloor d'_0 \rfloor \;\wedge\; d_1 = \lfloor d'_1 \rfloor : d_0 \sqsubseteq_D d_1)$$

The lifting construction $\lfloor - \rfloor : D \to D_\perp$ is continuous. We can extend lifting from cpos to continuous functions on cpos.

Definition. Let $f : D \to E$ be a continuous function from a cpo D to a cpo E with bottom $\perp_E$. We define the *extended* function $f^* : D_\perp \to E$ as

$$f^*(d') = \begin{cases} f(d) & \text{if } (\exists\, d \in D :: d' = \lfloor d \rfloor) \\ \perp_E & \text{otherwise} \end{cases}$$

Extending functions is a continuous operation, i.e. $(-)^*$ is a continuous function. With this we can define the *let*-construct.

Definition. Let D be a cpo and let x be a variable that ranges over the elements of D. Let E be a cpo with bottom. Let e be an expression, depending possibly on x, that is continuous and describes elements of E. That is, $\lambda x.e$ is a continuous function from D to E. When d' equals $\perp_D$ the result should be $\perp_E$, otherwise we want to substitute the underlying value of d' for x in e. Therefore we define

$$\mathit{let}\ x \Leftarrow d'.\, e \;\equiv\; (\lambda x.e)^*(d')$$

Multiple *let*-constructs may be abbreviated as one *let*-construct

$$\mathit{let}\ x_1 \Leftarrow c_1, x_2 \Leftarrow c_2, \ldots, x_n \Leftarrow c_n.\, e \;\equiv\; \mathit{let}\ x_1 \Leftarrow c_1.\, (\mathit{let}\ x_2 \Leftarrow c_2.\, (\cdots (\mathit{let}\ x_n \Leftarrow c_n.\, e) \cdots))$$

2.3.2 Products

Definition. Let D_1 and D_2 be cpos. We define their *product* $D_1 \times D_2$ to be the cpo with underlying set $D_1 \times D_2$ and coordinatewise ordering $\sqsubseteq_{D_1 \times D_2}$ defined as

$$(d_1, d_2) \sqsubseteq_{D_1 \times D_2} (d_1', d_2') \quad \equiv \quad d_1 \sqsubseteq_{D_1} d_1' \quad \wedge \quad d_2 \sqsubseteq_{D_2} d_2'$$

It is easy to check that this gives a cpo, and that upperbounds are taken coordinatewise, i.e.

$$\bigsqcup_{n \in \omega} (d_{1n}, d_{2n}) = (\bigsqcup_{n \in \omega} d_{1n}, \bigsqcup_{n \in \omega} d_{2n})$$

We associate *projection functions* $\pi_i : D_1 \times D_2 \to D_i$ (for $i = 1, 2$) with each product. The projection function π_i selects the ith coordinate from a tuple:

$$\pi_i(d_1, d_2) = d_i, \text{ with } i = 1, 2$$

The projection functions are continuous because least upperbounds are taken coordinatewise. This construction is easily generalized to a product of more than two cpos.

2.3.3 Sums

Definition. Let D_1 and D_2 be cpos. We define their *sum* $D_1 + D_2$ to be the cpo with underlying set

$$\{in_1(d_1) \mid d_1 \in D_1\} \cup \{in_2(d_2) \mid d_2 \in D_2\}$$

and partial order

$$\begin{array}{lll} d \sqsubseteq_{D_1 + D_2} d' & \equiv & (\exists\, d_1, d_1' \in D_1 : d = in_1(d_1) \ \wedge \ d' = in_1(d_1') : d_1 \sqsubseteq_{D_1} d_1') \ \vee \\ & & (\exists\, d_2, d_2' \in D_2 : d = in_2(d_2) \ \wedge \ d' = in_2(d_2') : d_2 \sqsubseteq_{D_2} d_2') \end{array}$$

where the *injection functions* in_1 and in_2 are injections that satisfy

$$(\forall\, d \in D_1, d' \in D_2 :: in_1(d) \neq in_2(d'))$$

The injection functions are continuous.

Definition. Let $D_1 + D_2$ be a sum of two cpos, and let E be a cpo. Let $f_1 : D_1 \to E$ and $f_2 : D_2 \to E$ be continuous functions. We can combine these two functions into the continuous function $[f_1, f_2] : D_1 + D_2 \to E$, which is defined as

$$[f_1, f_2](in_i(d_i)) = f_i(d_i) \quad \forall\, d_i \in D_i, \text{ with } i = 1, 2$$

Using this we can define the *case*-construct.

Definition. Let $d \in (D_1 + D_2)$ be an element in the sum of cpos D_1 and D_2. Let E be a cpo and assume that both $\lambda x_1.e_1 : D_1 \to E$ and $\lambda x_2.e_2 : D_2 \to E$ are continuous functions. We define the *case*-construct as:

$$\textit{case } d \textit{ of } in_1(x_1).e_1 \mid in_2(x_2).e_2 \quad \equiv \quad [\lambda x_1.e_1, \lambda x_2.e_2](d)$$

The set of truth values $\mathbb{B} = \{\textit{true}, \textit{false}\}$ can be regarded as the sum of the singleton cpos $\{\textit{true}\}$ and $\{\textit{false}\}$, with injection functions $in_1 : \{\textit{true}\} \to \mathbb{B}$ and $in_2 : \{\textit{false}\} \to \mathbb{B}$, two simple identity functions. A conditional can now be constructed as a *case*-construct that chooses an alternative depending on a truth value.

Definition. Let E be a cpo, $t \in \mathbb{B}$ a truth value and let $\lambda x_1.e_1 : \{\textit{true}\} \to E$ and $\lambda x_2.e_2 : \{\textit{false}\} \to E$ be two continuous functions. We define the simple conditional as

$$\textit{cond}(t, e_1, e_2) \quad \equiv \quad [\lambda x_1.e_1, \lambda x_2.e_2](t) \quad \equiv \quad \textit{case } t \textit{ of } in_1(x_1).e_1 \mid in_2(x_2).e_2$$

In the language Lager there is no Boolean datatype, so a second conditional based on integers is needed. Furthermore the value used to make a choice may be the result of a computation which may or may not return a result. In the latter case we say that the computation *diverges* or that it is a noncomputation. Diverging computations will be identified with the bottom-element of a cpo. So the choice is based on an element from the flat cpo $\mathbf{Z}_\perp$. We need an auxilliary continuous function $iszero : \mathbf{Z} \to \mathbb{B}$ that is defined as

$$iszero(n) = \begin{cases} true & \text{if } n = 0 \\ false & \text{otherwise} \end{cases}$$

This function is continuous because it is a function between discrete cpos.

Definition. Let $e_1, e_2 \in E$ for a cpo E with bottom and let $z_0 \in \mathbf{Z}_\perp$. We define an integer conditional as

$$Cond(z_0, e_1, e_2) \quad \equiv \quad let\ n \Leftarrow z_0\ .\ cond(iszero(n), e_1, e_2)$$

2.3.4 Function space

Definition. Let D and E be cpos. The sets of all continuous functions from D to E ordered pointwise forms a cpo, i.e. the *function space* $[D \to E]$ with underlying set

$$\{f \mid f : D \to E \quad \wedge \quad f \text{ is continuous}\}$$

and partial order

$$f \sqsubseteq_{[D \to E]} g \quad \equiv \quad (\forall\, d \in D :: f(d) \sqsubseteq_E g(d))$$

2.4 Information systems

In a denotational semantics each type will be associated with a cpo. The constructions above are sufficient if the types are nonrecursive. When the type equations are recursive, the specified type is the least solution to the type equation. There is need for a cpo of cpos wherein a least fixpoint can be taken. For this we will need information systems like those presented in chapter 12 of [Win92], which are not quite the same as the information systems that were invented by Dana Scott, but rather a special kind of *Scott*-domains. Another approach would have been to use category theory.

This section shall briefly touch upon information systems. The proof of numerous claims in the definitions are omitted and can be found in chapter 12 of [Win92].

Definition. An *information system* is a triple $\mathcal{A} = (A, \mathrm{Con}, \vdash)$, where

- A is a countable set (the tokens),
- Con (the consistent sets) is a nonempty subset of finite subsets of A and,
- $\vdash$ (the entailment relation) is a subset of $(\mathrm{Con} \setminus \{\emptyset\}) \times A$.

which satifies

1. $Y \in \mathrm{Con} \ \wedge \ X \subseteq Y \quad \Rightarrow \quad X \in \mathrm{Con}$
2. $a \in A \Rightarrow \{a\} \in \mathrm{Con}$
3. $X \vdash a \Rightarrow X \cup \{a\} \in \mathrm{Con}$
4. $X \in \mathrm{Con} \ \wedge \ a \in X \quad \Rightarrow \quad X \vdash a$
5. $X \in \mathrm{Con} \ \wedge \ Y \in \mathrm{Con} \ \wedge \ (\forall\, b \in Y :: X \vdash b) \ \wedge \ Y \vdash c \quad \Rightarrow \quad X \vdash c$

The tokens may be seen as the assertions that can be made of a computation. The set of consistent sets determines which tokens may hold simultaneously. The truth of some tokens may imply or entail the truth of of another token.

Definition. Let $\mathcal{A} = (A, \mathrm{Con}, \vdash)$ be an information system. The *elements* of $\mathcal{A}$ are those subsets x of A which are

1. *nonempty* : $x \neq \emptyset$
2. *consistent* : $X \subseteq x \ \wedge \ X$ is finite $\Rightarrow \ X \in \mathrm{Con}$
3. *entailment-closed* : $X \subseteq x \ \wedge \ X$ is finite $\wedge \ X \vdash a \ \Rightarrow \ a \in x$

We denote the set of elements of $\mathcal{A}$ by $|\mathcal{A}|$

An element is a nonempty, consistent subset of tokens. The consistent sets in Con are finite subsets of tokens, yet an element may be infinite. The infinite elements are constructed using the entailment relation.

Lemma. Let $\mathcal{A}$ be an information system, then the elements $|\mathcal{A}|$ ordered by inclusion form a cpo.

Example. We define the information system $\mathcal{Z}$ as

$$\mathcal{Z} = (\mathbb{Z}, \ \{\emptyset\} \cup \{\{n\} \mid n \in \mathbb{Z}\}, \ \{(\{n\}, n) \mid n \in \mathbb{Z}\})$$

It is easy to check that this is an information system and that the elements of $\mathcal{Z}$ form a cpo that is isomorphic to the cpo $\mathbf{Z}$

$$|\mathcal{Z}| = \{\{n\} \mid n \in \mathbb{Z}\} \cong \mathbb{Z}$$

The goal is to make a cpo of information systems so we need a partial order on information systems.

Definition. Let $\mathcal{A} = (A, \mathrm{Con}_A, \vdash_A)$ and $\mathcal{B} = (B, \mathrm{Con}_B, \vdash_B)$ be information systems. We define $\mathcal{A} \trianglelefteq \mathcal{B}$ iff

1. $A \subseteq B$
2. $X \in \mathrm{Con}_A \iff X \subseteq A \ \wedge \ X \in \mathrm{Con}_B$
3. $X \vdash_A a \iff X \subseteq A \ \wedge \ a \in A \wedge X \vdash_B a$

When $\mathcal{A} \trianglelefteq \mathcal{B}$ we say that $\mathcal{A}$ is a *subsystem* of $\mathcal{B}$. The subsystem relation is a partial order and has unique least element $\mathbf{0}$, the information system with no tokens and $\mathrm{Con} = \{\emptyset\}$.

We can now define the least upperbound of an ω-chain of information systems.

Definition. Let $\mathcal{A}_0 \trianglelefteq \mathcal{A}_1 \trianglelefteq \cdots \trianglelefteq \mathcal{A}_n \trianglelefteq \cdots$ be an ω-chain of information systems $\mathcal{A}_i = (A_i, \mathrm{Con}_i, \vdash_i)$. We construct the least upper bound of this chain as

$$\bigcup_i \mathcal{A}_i = (\bigcup_i A_i, \bigcup_i \mathrm{Con}_i, \bigcup_i \vdash_i)$$

The set of (all) information systems ordered by the subsystem relation forms a cpo. The definitions for monotone and continuous functions on cpos can be extended to information systems.

Definition. Let F be an operation on information systems. Then F is *monotone* iff for all information systems $\mathcal{A}$ and $\mathcal{B}$

$$\mathcal{A} \trianglelefteq \mathcal{B} \ \Rightarrow \ F(\mathcal{A}) \trianglelefteq F(\mathcal{B})$$

Let $\mathcal{A}_0 \trianglelefteq \mathcal{A}_1 \trianglelefteq \cdots \trianglelefteq \mathcal{A}_n \trianglelefteq \cdots$ be an ω-chain of information systems. Then F is *continuous* iff F is monotone and

$$\bigcup_i F(\mathcal{A}_i) = F(\bigcup_i \mathcal{A}_i)$$

The fixpoint theorem can also be extended to continuous operations on information systems. Recall that there is a least information system $\mathbf{0}$. This information system plays the role of the bottom element in the cpo of information systems.

Definition. If F is a continuous operation on information systems, then F has a least fixpoint $\mathit{fix}(F)$, defined as

$$\mathit{fix}(F) = \bigcup_i F^i(\mathbf{0})$$

which is the least upperbound of the ω-chain

$$\mathbf{0} \trianglelefteq F(\mathbf{0}) \trianglelefteq F(F(\mathbf{0})) \trianglelefteq \cdots \trianglelefteq F^n(\mathbf{0}) \trianglelefteq \cdots$$

2.5 Constructions on information systems

The previous constructions on cpos like lifting, product and sum can also be extended to information systems. We are not interested in the exact details of the operation which can be found in chapter 12 of [Win92].

2.5.1 Lifting

Lifting on information systems will have the effect that the underlying cpo is lifted.

Definition. Let $\mathcal{A}$ be an information system. Then $\mathcal{A}_\perp$ is the lifted information system. The elements of both information systems can be related as

$$y \in |\mathcal{A}_\perp| \iff y = \{\emptyset\} \vee (\exists\, x \in |\mathcal{A}| :: y = \{b \mid b \subseteq^{\text{fin}} x\})$$

The cpos $|\mathcal{A}_\perp|$ and $|\mathcal{A}|_\perp$ are isomorphic, i.e. $|\mathcal{A}_\perp| \cong |\mathcal{A}|_\perp$ with the mapping

$$x \mapsto \begin{cases} \lfloor \bigcup x \rfloor & \text{if } x \neq \{\emptyset\} \\ \perp_{|\mathcal{A}|} & \text{if } x = \{\emptyset\} \end{cases}$$

Lifting on information systems is, like lifting on cpos, a continuous operation. The associated lifting function $\lfloor - \rfloor : |\mathcal{V}| \to |\mathcal{V}_\perp|$ will be taken as

$$\lfloor x \rfloor = \{b \mid b \subseteq^{\text{fin}} x\}$$

and the bottom element as

$$\perp = \{\emptyset\}$$

We shall not use explicit ismorphisms between isomorphic cpos, but rather leave them out for simplicity. That is, if $x \in |\mathcal{V}|$ then $\lfloor x \rfloor$ is an element of $|\mathcal{V}_\perp|$ and also of $|\mathcal{V}_\perp|$ because of the isomorphism.

The $(-)^*$ operation on cpos can now be defined for information systems, but because we assume that isomorphic cpos are equal (which they are except for a "renaming" of elements) we can get away with the old definition. Consequently the old definition of *let* also suffices. The same goes for the definitions of *case* and *cond*. The old definition of the integer conditional is also sufficient if we observe the isomorphism between $|\mathcal{Z}|$ and $\mathbf{Z}$, which is given by $\{n\} \mapsto n$.

2.5.2 Products

Definition. Let $\mathcal{A}$ and $\mathcal{B}$ be information systems, then $\mathcal{A} \times \mathcal{B}$ is the product of both information systems. The elements of $\mathcal{A}, \mathcal{B}$ and $\mathcal{A} \times \mathcal{B}$ are related as

$$x \in |\mathcal{A} \times \mathcal{B}| \Longleftrightarrow (\exists\, x_1 \in |\mathcal{A}|, x_2 \in |\mathcal{B}| :: x = x_1 \times x_2)$$

There is an isomorphism from $|\mathcal{A} \times \mathcal{B}|$ to $|\mathcal{A}| \times |\mathcal{B}|$ given by $(x_1, x_2) \mapsto x_1 \times x_2$. Thus

$$|\mathcal{A} \times \mathcal{B}| \cong |\mathcal{A}| \times |\mathcal{B}|$$

The product on information systems is a continuous operation.

2.5.3 Sums

Definition. Let $\mathcal{A}$ and $\mathcal{B}$ be information systems, then $\mathcal{A} + \mathcal{B}$ is the sum of both information systems. The elements of $\mathcal{A}, \mathcal{B}$ and $\mathcal{A} + \mathcal{B}$ are related as

$$x \in |\mathcal{A} + \mathcal{B}| \Longleftrightarrow (\exists\, y \in |\mathcal{A}| :: x = inj_1 y) \vee (\exists\, y \in |\mathcal{B}| :: x = inj_2 y)$$

Where $inj_1 : A \to A \uplus B$ and $inj_2 : B \to A \uplus B$ are defined as $inj_1 : a \mapsto (1, a)$ for $a \in A$ and $inj_2 : b \mapsto (2, b)$ for $b \in B$. There is an isomorphism of the cpos $|\mathcal{A} + \mathcal{B}|$ and $|\mathcal{A}| + |\mathcal{B}|$ given by

$$x \mapsto \begin{cases} in_1(y) & \text{if } x = inj_1 y \\ in_2(y) & \text{if } x = inj_2 y \end{cases}$$

Thus

$$|\mathcal{A} + \mathcal{B}| \cong |\mathcal{A}| + |\mathcal{B}|$$

Summing information systems is a continuous operation.

2.5.4 Lifted function space

The general function space construction can not be defined on information systems. However, the function space with a lifted range can.

Definition. Let $\mathcal{A}$ and $\mathcal{B}$ be information systems. Then the *lifted function space* is denoted by $\mathcal{A} \to \mathcal{B}_\perp$. There is an isomorphism between $|\mathcal{A} \to \mathcal{B}_\perp|$ and $[|\mathcal{A}| \to |\mathcal{B}_\perp|]$. However, we need a slightly different lifted function space where both domain and range are lifted. There is again an isomorphism, this time between $|\mathcal{A}_\perp \to \mathcal{B}_\perp|$ and $[|\mathcal{A}_\perp| \to |\mathcal{B}_\perp|]$. This latter isomorphism is also intricate and therefore the details are not included here.

2.6 Domain equations

With the above theory we can now solve (recursive) domain equations of the following form

$$\mathcal{X} = F(\mathcal{X})$$

where $\mathcal{X}$ is an information system and F is a continuous operation on information systems built up from the continuous operations given in the previous section and the basic information systems $\mathbf{0}$ and $\mathcal{Z}$. We shall take the solution of the above equation to be

$$fix(F)$$

which yields not a precise equality but rather an isomorphic equality of the underlying cpos, i.e.

$$|fix(F)| \cong |F(fix(F))|$$

Because we regard isomorphic cpos as equal, this is sufficient. The cpo assigned to a recursive type, is the underlying cpo of the least information system that satifies the domain equation which introduces the recursive type.

Chapter 3

Lapa

3.1 Introduction to Lapa

Lapa is a simple functional language that can be studied and implemented by undergraduate students. Lapa is kept simple to allow possible formal verification of the operational semantics by means of a mechanical theorem prover.

Operators consist of single characters to facilitate the parsing of a Lapa program. There is one overloaded operater . which is used for both function application and pair selection. Pairs come in two flavours, lazy and eager. The difference between lazy and eager is made with pair selection.

Lapa can be untyped or typed. In the latter case each function-variable, variable and symbolic constant must be given a type, and every term has to be well-typed. The type system allows definition of recursive types, like lists. It also allows for the definition of non recursive types or type aliases. The only base type is the type of integers. Booleans can be encoded as integers by letting 0 represent *true* and every other nonzero integer represent *false*. Note that this is the exact opposite of the convention used in the programming language C.

The goal is to specify a denotational semantics for Lapa, if possible. In the denotational semantics a program(fragment) is assigned a meaning by associating it with an element from a domain of possible meanings. This domain is dependent on the type of the program, and thus enough type information is needed to determine the proper domain for a given program fragment. We shall therefore present typed Lapa, which is effectively untyped Lapa with some type restrictions.

3.2 The definition of typed Lapa

3.2.1 Program structure

A Lapa program consists of some type definitions intermixed with type declarations of variables, symbolic constants and/or function variables. After the type definitions and declarations the functions are defined. The last item in a Lapa program is a (well-typed) term that will be evaluated under the definitions in the program. This can be stated somewhat more formal as:

Definition. A Lapa program is a triple

```
Lapa program  = (  type definitions and declarations,
                   function definitions,
                   a single Lapa term   )
```

In an interactive setting the type definitions, type declarations and function definitions may form a script that is given to an interpreter. The Lapa term is the term one enters in a session at the interpreter prompt. More than one Lapa term may then be given for evaluation.

The definition can be extended to include more than one Lapa term, but for simplicity only a single term is considered.

For the sake of presentation we shall first take a look at the terms and function definitions, then look at the operational semantics and after that we dive into the type system and state some problems with Lapa.

3.2.2 Abstract syntax of Lapa terms

We are not interested in the precise concrete syntax of a Lapa program, but rather in an abstract syntax which captures the essence of a Lapa term.

We define a syntactic set **Term** of terms by means of an inductive definition. To this end we assume there are already four syntactically discernable sets with a notation convention between parentheses:

- **Num** the set of integers ($n, n_0, n_1, \ldots \in$ **Num**),
- **Var** an infinite set of variables ($x, x_0, x_1, \ldots \in$ **Var**),
- **FVar** an infinite set of function variables ($f, f_0, f_1, \ldots \in$ **FVar**),
- **Sym** an infinite set of symbolic constants ($s, s_0, s_1, \ldots \in$ **Sym**).

Symbolic constants in Lapa are regular identifiers preceded by a single quote. This choice facilitates a possible translation of a Lapa term into an expression in the Boyer-Moore theorem prover NQTHM, which has quoted constants.

Next we define **Term**, for which we adopt the following notation convention: $t, t_0, t_1, \ldots \in$ **Term**. A term t is built up according to the following abstract syntax:

t	::=	n	integer
	|	s	symbolic constant
	|	x	variable
	|	t_1 + t_2	arithmetic
	|	t_1 - t_2	
	|	t_1 * t_2	
	|	t_1 / t_2	
	|	t_1 % t_2	
	|	t_1 < t_2	relational
	|	t_1 = t_2	
	|	(t_1 ; t_2)	lazy pair
	|	(t_1 , t_2)	eager pair
	|	t_1 . t_2	application, pair selection
	|	x : t_1	abstraction
	|	f	function variable

For the abstract syntax we could have chosen to remove the overloading of the application operator by introducing yet another operator specifically for pair selection. Note that : is used for both type expressions and abstractions (and later also for function definitions), which may seem somewhat unconventional.

A function is defined by assigning a term to a function variable. The defined function should only be dependent on the expressions that will be substituted for variables introduced by abstractions. In other words, the defining term should be *closed*. To formalize this we define a function FV that returns the free variables, which are those variables that are not in the scope of an abstraction.

$\mathrm{FV}(n) = \mathrm{FV}(s) = \mathrm{FV}(f) = \emptyset$,
$\mathrm{FV}(x) = \{x\}$,
$\mathrm{FV}(t_1 \ \mathbf{op} \ t_2) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$ with $\mathbf{op} = +, -, *, /, \%, <, =$,
$\mathrm{FV}((t_1 \ ; \ t_2)) = \mathrm{FV}((t_1 \ , \ t_2)) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$

$\mathrm{FV}(t_1 \,.\, t_2) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$
$\mathrm{FV}(x : t) = \mathrm{FV}(t) \setminus \{x\}$

Definition. A term $t \in \mathbf{Term}$ is *closed* iff $\mathrm{FV}(t) = \emptyset$

Definition. A *sequence of function definitions* consists of zero or more equations between function variables and closed terms:

$$\begin{array}{lcl} f_1 & : & t_1 \\ & \vdots & \\ f_k & : & t_k \end{array}$$

for a natural number k. We require that a function variable is defined at most once, i.e. $f_i \neq f_j$ for $1 \leq i < j \leq k$.

It is not required that each function variable on the right-hand side of a function definition should occur somewhere on the left-hand side of another function definition. Those function variables are not defined and cannot not be rewritten.

For the operational semantics we shall assume that the equations above are the function definitions that are given.

3.2.3 An operational semantics

For an operational semantics for Lapa, we need to know what its normal forms are. The normal forms of Lapa cannot be characterized without the rewrite rules. A term is in normal form when it is closed and cannot be rewritten further. We want to define, by means of rules, a rewrite relation $\longrightarrow$ that rewrites a term to its normal form. Rewriting a term will then correspond to building a proof tree using the rules. The only problem is that rules for normal forms need to be given to complete the proof trees. But closed terms that cannot be rewritten are in normal forms, so proof rules that state that those terms are in normal form need to be added. But this is not possible, as we could otherwise enumerate the normal forms without the rules. We need to bend the rules a bit and rely on a sort of specificity order in which to apply the rules. The most specific rules apply first, and a general rewrite rule is added to allow completion of proof trees. All rules except this general rule will have mutual exclusive premisses. (We denote terms in normal forms with: $c, c_0, c_1, \ldots$)

The rewrite rules

integers:

$$\frac{}{n \longrightarrow n}$$

symbolic constants:

$$\frac{}{s \longrightarrow s}$$

arithmetic:

$$\frac{t_1 \longrightarrow n_1 \;\wedge\; t_2 \longrightarrow n_2}{t_1 \;\mathbf{op}\; t_2 \longrightarrow (n_1 \mathit{op}\, n_2)} \quad \text{with} \quad \begin{array}{lcl} \mathbf{op} & = & +,\ -,\ * \\ \mathit{op} & = & \text{mathematical version of } \mathbf{op} \end{array}$$

$$\frac{t_1 \longrightarrow n_1 \;\wedge\; t_2 \longrightarrow n_2 \;\wedge\; n_2 \not\equiv 0}{t_1 \;\mathbf{op}\; t_2 \longrightarrow (n_1 \mathit{op}\, n_2)} \quad \text{with} \quad \begin{array}{lcl} \mathbf{op} & = & /,\ \% \\ \mathit{op} & = & \text{mathematical version of } \mathbf{op} \end{array}$$

relational:

$$\frac{t_1 \longrightarrow n_1 \;\wedge\; t_2 \longrightarrow n_2 \;\wedge\; n_1 < n_2}{t_1 < t_2 \longrightarrow 0} \qquad \frac{t_1 \longrightarrow n_1 \;\wedge\; t_2 \longrightarrow n_2 \;\wedge\; n_2 \leqslant n_1}{t_1 < t_2 \longrightarrow 1}$$

$$\frac{t_1 \longrightarrow n_1 \;\wedge\; t_2 \longrightarrow n_2 \;\wedge\; n_1 \equiv n_2}{t_1 = t_2 \longrightarrow 0} \qquad \frac{t_1 \longrightarrow n_1 \;\wedge\; t_2 \longrightarrow n_2 \;\wedge\; n_1 \not\equiv n_2}{t_1 = t_2 \longrightarrow 1}$$

lazy pairs:
$$\frac{\mathrm{FV}(t_1) = \mathrm{FV}(t_2) = \emptyset}{(t_1\,;\,t_2) \longrightarrow (t_1\,;\,t_2)}$$

eager pairs:
$$\frac{t_1 \longrightarrow c_1 \ \wedge \ t_2 \longrightarrow c_2}{(t_1\,,\,t_2) \longrightarrow (c_1\,,\,c_2)}$$

application:
$$\frac{t_1 \longrightarrow x : t_1' \ \wedge \ t_1'[t_2/x] \longrightarrow c}{t_1\,.\,t_2 \longrightarrow c}$$

lazy pair selection:
$$\frac{t_1 \longrightarrow n \ \wedge \ t_2 \longrightarrow (t_{21}\,;\,t_{22}) \ \wedge \ n \equiv 0 \ \wedge \ t_{21} \longrightarrow c}{t_1\,.\,t_2 \longrightarrow c}$$

$$\frac{t_1 \longrightarrow n \ \wedge \ t_2 \longrightarrow (t_{21}\,;\,t_{22}) \ \wedge \ n \not\equiv 0 \ \wedge \ t_{22} \longrightarrow c}{t_1\,.\,t_2 \longrightarrow c}$$

eager pair selection:
$$\frac{t_1 \longrightarrow n \ \wedge \ t_2 \longrightarrow (c_1\,,\,c_2) \ \wedge \ n \equiv 0}{t_1\,.\,t_2 \longrightarrow c_1}$$

$$\frac{t_1 \longrightarrow n \ \wedge \ t_2 \longrightarrow (c_1\,,\,c_2) \ \wedge \ n \not\equiv 0}{t_1\,.\,t_2 \longrightarrow c_2}$$

abstractions:
$$\frac{\mathrm{FV}(x : t) = \emptyset}{x : t \longrightarrow x : t}$$

function variables:
$$\frac{f \text{ is not defined}}{f \longrightarrow f}$$

$$\frac{f = t \text{ is given as a definition} \quad t \longrightarrow c}{f \longrightarrow c}$$

force normal forms:
$$\frac{\mathrm{FV}(t) = \emptyset}{t \longrightarrow t}$$

Note that the rules for integers, symbolic constants, lazy pairs, abstractions and undefined function variables are special instances of the last rule.

Example definitions

```
From : (n : (n  ; (From . (n + 1))))
Take : (n : (xs : ((n = 0) . ('Nil ; ((0 . xs) , ((Take . (n - 1)) . (1 . xs)))))))
Sum : (xs :  (xs = 'Nil) . (0 ; ( (0 . xs) + (Sum . (1 . xs)))))
```

Computation of `From.4` yields `(4 ; (From . (4 + 1)))` and computation of `(Take.2).(From.7)` yields `(7 , (8 , 'Nil))` which can both be easily verified by building proof trees. Computation of `Sum.((Take.3).(From.1))` yields 6. The typing declarations will be given later.

Remark. The operational semantics given here differs a bit from the original operational semantics of Lapa which computed results while performing a substitution. The computation of `From.4` under the original semantics yielded `(4 ; (From . 5)))`.

3.2.4 Type definitions and declarations

Lapa has only one base type, **Z** the type of the integers. As we have already observed truth values or booleans are represented by integers. With the type definitions new, possibly recursive, types can be introduced. Type aliases can also be given.

New types or type aliases are defined by assigning a type expression to a type variable. We assume there is an infinite set of syntactic entities, the set **TVar** of type variables. We implicitely assume that the entities from this set are distinct from all elements of other sets we may yet introduce.

We define the syntactiç set **TExp** of type expressions through an inductive definition

$$
\begin{array}{lcll}
\tau & ::= & \mathbf{Z} & \text{integers} \\
 & | & \tau_1 \, , \tau_2 & \text{pairs} \\
 & | & \tau_1 : \tau_2 & \text{function} \\
 & | & X & \text{type variable}
\end{array}
$$

with $X \in \mathbf{TVar}$ and $\tau, \tau_1, \tau_2 \in \mathbf{TExp}$.

Definition. A *type definition* is an assignment $X = \tau$ of a type expression $\tau \in \mathbf{TExp}$ to a type variabele $X \in \mathbf{TVar}$. We require that each type variable used on the right-hand side of a type definition occurs precisely once on the left-hand side of another, possibly the same, type definition.

Definition. A *type declaration* is an assignment of the form $v \mathbin{@} \tau$, where v is either a symbolic constant, a variable or a function variable and τ is the type expression which will be the type associated with v.

To avoid possible ambiguities we require that each type expression on the right-hand side of a type declaration contains only type variables that are defined, i.e. type variables that occur exactly once on the left-hand side of some type definition. Furthermore the same symbolic constant, variable or function variable may occur at most once on the left-hand side of a type declaration.

Some of the type declarations may be seen as part of a type definition, in particular type declarations that assign a type to a symbolic constant. This is needed to introduce the base elements of a recursive type. Lists, for example, can be implemented as pairs where the first component is an element, or more precise by the head of the list and the second component the remaining list or tail. So we can define the type of lists of integers as recursively a recursive type

```
Zlist = Z, Zlist
```

But the empty list would then also be a pair. To solve this a declaration of a symbolic constant `'Nil` of type `Zlist` is given. This symbolic constant will then represent the the empty list. Thus the *complete* type definition of the type `Zlist` is

```
Zlist = Z, Zlist
'Nil @ Zlist
```

With this type definition and type declaration we complete the type declarations for the example function definitions:

```
n @ Z
xs @ Zlist
From @ (Z : Zlist)
Take @ (Z : (Zlist : Zlist))
Sum @ (Zlist : Z)
```

3.2.5 Typing rules

To check the types of terms, typing rules are needed. Unfortunately Lapa contains a flaw in the typing, and therefore we are not able to complete the typing rule for pair selection. We now present the typing rules for as far as possible and in the next section we shall discuss why pair selection is untypable.

We assume that $type : \mathbf{Var} \cup \mathbf{FVar} \cup \mathbf{Sym} \to \mathbf{TExp}$ is a function that represents the typing information given in the type declarations.

integers: $$\frac{}{n : \mathbb{Z}}$$

symbolic constants: $$\frac{type(s) = \tau}{s \mathbin{@} \tau}$$

variables: $$\frac{type(x) = \tau}{x \mathbin{@} \tau}$$

arithmetic: $$\frac{t_1 \mathbin{@} \mathbb{Z} \quad t_2 \mathbin{@} \mathbb{Z}}{t_1 \ \mathbf{op} \ t_2 \mathbin{@} \mathbb{Z}}$$ with **op** = +,-,*,/,%

relational: $$\frac{t_1 \mathbin{@} \mathbb{Z} \quad t_2 \mathbin{@} \mathbb{Z}}{t_1 < t_2 \mathbin{@} \mathbb{Z}}$$

$$\frac{t_1 \mathbin{@} \tau \quad t_2 \mathbin{@} \tau}{t_1 = t_2 \mathbin{@} \mathbb{Z}}$$

pairs: $$\frac{t_1 \mathbin{@} \tau_1 \quad t_2 \mathbin{@} \tau_2}{(t_1 \ ; \ t_2) \mathbin{@} \tau_1, \tau_2}$$

$$\frac{t_1 \mathbin{@} \tau_1 \quad t_2 \mathbin{@} \tau_2}{(t_1 \ , \ t_2) \mathbin{@} \tau_1, \tau_2}$$

application: $$\frac{t_1 \mathbin{@} \tau_1 : \tau_2 \quad t_2 \mathbin{@} \tau_1}{t_1 \ . \ t_2 \mathbin{@} \tau_2}$$

abstraction: $$\frac{x \mathbin{@} \tau_1 \quad t \mathbin{@} \tau_2}{x : t \mathbin{@} \tau_1 : \tau_2}$$

function variable: $$\frac{type(f) = \tau}{f \mathbin{@} \tau}$$

pair selection: $$\frac{t_1 \mathbin{@} \tau_1 \quad t_2 \mathbin{@} \tau_1, \tau_2}{t_1 \ . \ t_2 \mathbin{@} ????}$$

3.3 Problems with Lapa

The pair selection cannot be typed within the type system, because the components may have different types and there is no way to express a combination of both types in the type system. Special operations that select the first or the second component of a pair could be added with correct typing rules. But the value used for pair selection is not always known at compile time, so a conditional would be needed to select the appropriate component selector based on this value. This conditional could yield either branch and so the same typing problems arises. The logical typing rule for a conditional would be to require both branches to be of the same type, but then the conditional could not be used for the problem above.

To repair this problem, the type system should be extended with a sum type. A value in a sum type is either a member of the type on the left-side of the sum or a member of the type on the right-side of the sum. Adding a new type is not enough, new statements to manipulate sum types are also needed.

The inability to type pair selection is not the only problem. The typing rules also allow equality on function types, which in principle is undecidable. Thus equality on function types should not be allowed, for it is not implementable

Apart from the exclusion of function types, the restriction on the types that can be compared for equality is still quite involved. For recursive types Lapa relies on the ability to compare symbolic constants for equality. Take a look at the definition of `Sum`, which uses the comparison `(xs = 'Nil)` to check whether the list of integers `xs` is empty or not. Suppose we were to declare

:he following symbolic constant

```
'Nel @ (Z, Zlist)
```

Then `'Nel` would also be a member of the type of lists of integers, because the type of lists of integers is defined as being a pair with the first component an integer and the second component a list of integers. But `'Nel` could also be a pair with the first component is an integer and the second component is a list. We are unable to truly distinguish a pairs from recursive types like lists.

To solve this recursive types should not be represented as pairs. This involves modifying the type system to allow definition of recursive types, and adding language constructs for dealing with recursive types.

Yet another problem, which has already been observed, is that some type declarations extend the type by introducing new elements.

Due to these deficiencies we decided to drop Lapa and construct a language with roughly the same features. The result is Lager, which stands for Lazy and Eager.

Chapter 4

Lager

4.1 Introduction to Lager

Lager is based largely on the lazy and eager functional languages described in [Win92]. Most language constructs are taken from the lazy language presented there in chapter 11. A base type of integers has been added and an eager `let` construct has been introduced to allow some control over the evaluation order. This construct is taken to be eager and can be used to control the evaluation order of terms which was controlled in Lapa by means of eager lists. But the construct here is somewhat more flexible.

4.2 Program structure

Like a Lapa program, a Lager program consists of some type definitions and declarations followed by a sequence of function definitions and a term that needs to be evaluated with respect to the definitions. With the type definitions one can introduce new, possibly recursive, types. With the type declarations type information of variables and functions can be introduced. Contrary to Lapa, type declarations will not form part of the definition of a type, i.e. will not introduce extra elements into a type.

4.3 The type system of Lager

In Lager there are two base types: the empty type 0 which has no elements, and the type of integers Z. Just like Lapa, booleans can be represented as integers, i.e. *true* can be represented as the integer 0, and *false* can be represented as any nonzero integer, the exact opposite of the convention in the language C.

New types can be constructed by means of a sequence of type definitions, which are equations between type variables and type expressions. We assume there is an infinite syntactic set **TVar** of type variables, with entities distinct from all the entities of other sets that we will introduce. By means of an inductive definition we introduce the syntactic set of **TExp** of type expressions:

τ	::=	0	empty type
	|	Z	the integer type
	|	τ_1 * τ_2	product
	|	τ_1 + τ_2	sum
	|	τ_1 -> τ_2	function type
	|	X	type variable

with $X \in$ **TVar** and $\tau, \tau_1, \tau_2 \in$ **TExp**.

We assume the following notation convention

$$\tau, \tau_0, \tau_1, \ldots \in \mathbf{TExp}$$

A type definition is an assignment of a type expression to a type variable. This is generalized to a sequence of type definitions which has the following form:

$$\begin{aligned} T_1 &= u_1 \\ &\vdots \\ T_m &= u_m \end{aligned}$$

for a natural number m, with $T_1, \ldots, T_m \in \mathbf{TVar}$ and $u_1, \ldots, u_m \in \mathbf{TExp}$.

For the remaining chapter we will assume that these equations are the type definitions that are given.

We further require that the sequence of type definitions is closed, i.e. each type variable on the right-hand side of a definition is defined itself in the type definitions. Formalisation of this requirement needs a function FTV which gives the (free) type variables of a type expression:

$\mathrm{FTV}(0) = \mathrm{FTV}(\mathtt{Z}) = \emptyset$
$\mathrm{FTV}(\tau_1 \mathtt{*} \tau_2) = \mathrm{FTV}(\tau_1 \mathtt{+} \tau_2) = \mathrm{FTV}(\tau_1 \mathtt{->} \tau_2) = \mathrm{FTV}(\tau_1) \cup \mathrm{FTV}(\tau_2)$
$\mathrm{FTV}(X) = \{X\}$

with $X \in \mathbf{TVar}$.

We also need to know whether a type is recursive or nonrecursive, i.e. just a type synonym. To differentiate between these alternatives, we only allow a type to be recursively defined in itself. Mutual recursion between type variables is not allowed (except for simple recursion). A type associated with type variable T_i is now *recursive* if T_i occurs as a (free) type variable in u_i, that is $T_i \in \mathrm{FTV}(u_i)$.

So the requirement that the sequence of type definitions is closed together with the previous requirement can formalized as:

$$(\forall i : 1 \leq i \leq m : \mathrm{FTV}(u_i) \subseteq \{T_1, \ldots, T_i\})$$

Example. The type `Zlist` of list of integers can now be defined as follows

```
Zlist = 0 + (Z * Zlist)
```

This might seems equal to

```
Zlist = Z * Zlist
```

but then `Zlist` would be the type of infinite and partial lists of integers and would not include the finite lists of integers. The first definition states that a list of type `Zlist` either 'empty' is or a pair. In section 4.9 information systems and thus effectively cpos will be assigned to user defined types.

4.4 Abstract syntax of terms

A function is defined by assigning a term to a function variable. We therefore define the abstract syntax of terms before tackling the function definitions. We assume there are three distinct syntactical sets with a notation convention between parentheses:

- **Num** the set of integers ($n, n_0, n_1, \ldots \in \mathbf{Num}$),
- **Var** an infinite set of variables ($x, x_0, x_1, \ldots \in \mathbf{Var}$)
- **FVar** an infinite set of function variables ($f, f_0, f_1, \ldots \in \mathbf{FVar}$).

The syntactic set of terms **Term**, with notation convention $t, t_0, t_1, \ldots \in$ **Term**, is inductively defined as:

t	::=	$\bullet$	diverging computation
		n	integer
		x	variable
		t_1 + t_2	arithmetic
		t_1 - t_2	
		t_1 * t_2	
		t_1 / t_2	
		t_1 % t_2	
		t_1 < t_2	relational
		t_1 = t_2	
		if t_0 then t_1 else t_2	conditional
		(t_1 , t_2)	pair
		fst(t_1)	operations on pairs
		snd(t_1)	
		inl(t_1)	constructs for sum types
		inr(t_1)	
		case t_0 of inl(x_1) . t_1 , inr(x_2) . t_2	case statement
		let x_1 <= t_1 in t_2	eager let statement
		t_1 . t_2	application
		$\lambda\, x\, .\, t_1$	abstraction
		f	function variable
		abs(t_1)	operations on recursive types
		rep(t_1)	

4.5 Typing Rules

We can now present the typing rules for terms. The typing rules are partly dependent on the type definitions and declarations that are given in the Lager program. We assume that the type of each variable and each function variable is given in the type declarations. That is, we assume a function $type$: **Var** $\cup$ **FVar** $\rightarrow$ **TExp** is given which assigns a type to each variable or function variable.

Diverging computation:
$$\frac{}{\bullet : \mathbf{0}}$$

Integers:
$$\frac{}{n : \mathtt{Z}}$$

Variables:
$$\frac{type(x) = \tau}{x : \tau}$$

Arithmetic and relational:
$$\frac{t_1 : \mathtt{Z} \quad t_2 : \mathtt{Z}}{t_1 \ \mathbf{op}\ t_2 : \mathtt{Z}} \qquad \text{with } \mathbf{op} = +, -, *, /, \%, <, =,$$

Conditional:
$$\frac{t_0 : \mathtt{Z} \quad t_1 : \tau \quad t_2 : \tau}{\mathtt{if}\ t_0\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 : \tau}$$

Products:
$$\frac{t_1 : \tau_1 \quad t_2 : \tau_2}{(t_1\,,\, t_2) : \tau_1 * \tau_2} \qquad \frac{t : \tau_1 * \tau_2}{\mathtt{fst}(t) : \tau_1} \qquad \frac{t : \tau_1 * \tau_2}{\mathtt{snd}(t) : \tau_2}$$

Sums:

$$\frac{t : \tau_1}{\texttt{inl}(t) : \tau_1 + \tau_2} \qquad \frac{t : \tau_2}{\texttt{inr}(t) : \tau_1 + \tau_2}$$

$$\frac{t_0 : \tau_1 + \tau_2 \quad x_1 : \tau_1 \quad t_1 : \tau \quad x_2 : \tau_2 \quad t_2 : \tau}{\texttt{case } t_0 \texttt{ of inl}(x_1).\, t_1,\ \texttt{inr}(x_2).\, t_2 : \tau}$$

Let statement:

$$\frac{x_1 : \tau_1 \quad t_1 : \tau_1 \quad t_2 : \tau}{\texttt{let } x_1 \texttt{<= } t_1 \texttt{ in } t_2 : \tau}$$

Function types:

$$\frac{x_1 : \tau_1 \quad x_2 : \tau_2}{\lambda x_1.\, t_1 : \tau_1 \texttt{ -> } \tau_2} \qquad \frac{t_1 : \tau_1 \texttt{ -> } \tau_2 \quad t_2 : \tau_1}{t_1\ .\ t_2 : \tau_2}$$

$$\frac{\mathit{type}(f) = \tau}{f : \tau}$$

Recursive types:

$$\frac{t : T_i \quad T_i = u_i \text{ (recursive type definition)}}{\texttt{rep}(t) : u_i}$$

$$\frac{t : u_i \quad T_i = u_i \text{ (recursive type definition)}}{\texttt{abs}(t) : T_i}$$

Non-recursive types:

$$\frac{t : T_i \quad T_i = u_i \text{ (type synonym)}}{t : u_i}$$

$$\frac{t : u_i \quad T_i = u_i \text{ (type synonym)}}{t : T_i}$$

Only the last four typing rules are dependent of the type definitions, and show the need for the distinction between recursive and nonrecursive types. The operator `rep` serves to unfold a recursive type to its recursive definition. Such unfolding cannot or need not be done to nonrecursive types.

4.6 Function definitions

We first introduce a function FV that returns the free variables of a given term:

$\mathrm{FV}(\bullet) = \mathrm{FV}(n) = \mathrm{FV}(f) = \emptyset$,
$\mathrm{FV}(x) = \{x\}$,
$\mathrm{FV}(t_1 \texttt{ op } t_2) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$, op = +,-,*,/,%,<,=,
$\mathrm{FV}((t_1, t_2)) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$
$\mathrm{FV}(\texttt{fst}(t_1)) = \mathrm{FV}(\texttt{snd}(t_1)) = \mathrm{FV}(t_1)$
$\mathrm{FV}(\texttt{if } t_0 \texttt{ then } t_1 \texttt{ else } t_2) = \mathrm{FV}(t_0) \cup \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$
$\mathrm{FV}(\texttt{inl}(t_1)) = \mathrm{FV}(\texttt{inr}(t_1)) = \mathrm{FV}(t_1)$
$\mathrm{FV}(\texttt{case } t_0 \texttt{ of inl}(x_1).t_1,\ \texttt{inr}(x_2).t_2) = \mathrm{FV}(t_0) \cup (\ \mathrm{FV}(t_1) \setminus \{x_1\}\ ) \cup (\ \mathrm{FV}(t_2) \setminus \{x_2\}\ )$
$\mathrm{FV}(\texttt{let } x_1 \texttt{ <= } t_1 \texttt{ in } t_2) = \mathrm{FV}(t_1) \cup (\ \mathrm{FV}(t_2) \setminus \{x_1\}\ )$
$\mathrm{FV}(t_1\ .\ t_2) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$
$\mathrm{FV}(\lambda x_1.t_1) = \mathrm{FV}(t_1) \setminus \{x_1\}$
$\mathrm{FV}(\texttt{abs}(t_1)) = \mathrm{FV}(\texttt{rep}(t_1)) = \mathrm{FV}(t_1)$

Definition. A term $t \in$ **Term** is *closed* iff $\mathrm{FV}(t) = \emptyset$.

Definition. The *function definitions* form a sequence of equations between distinct function variables and closed terms:

$$\begin{aligned} f_1 &= d_1 \\ &\vdots \\ f_k &= d_k \end{aligned}$$

for a natural number k, with $f_1, \ldots, f_k \in$ **FVar**, $f_i \neq f_j$ for $1 \leq i < j \leq k$, and $d_1, \ldots, d_k \in$ **Term**.

We not only require that the defining terms d_i are closed, but also that each function variable f_i is defined only once and that all function variables on the right-hand sides of the equations appear on the left-hand side of some equation.

For the remainder of the chapter we will assume that the above definition are the given function definitions.

Example. The functions From, Take and Sum previously given for Lapa can be defined in Lager including type declarations as

```
n : Z
xs : Zlist
l : 0
r : Z * Zlist
From : Z -> Zlist
Take : Z -> (Zlist -> Zlist)
Sum : Zlist -> Z

From = \n . abs(inr( (n, From.(n + 1)) ))
Take = \n . \xs . if n then abs(inl(@))
                      else case rep(xs) of
                             inl(l) . abs(inl(@)),
                             inr(r) . abs(inr( (fst(r), (Take . (n-1)) . (snd(r))) ))
Sum = \xs . case rep(xs) of
              inl(l) . 0,
              inr(r) . fst(r) + (Sum . (snd(r)))
```

We write a backslash for λ and @ for $\bullet$.

4.7 Normal forms and head normal forms

Before an operational semantics can be given the normal forms need to be specified. We define normal forms per closed type as a set, i.e.

N_τ = the set of normal forms for closed type τ

by means of the following rules

Integers:
$$\frac{n : \mathsf{Z}}{n \in N_{\mathsf{Z}}}$$

Pairs:
$$\frac{c_1 \in N_{\tau_1} \quad c_2 \in N_{\tau_2}}{(c_1, c_2) \in N_{\tau_1 * \tau_2}}$$

Sums:
$$\frac{c \in N_{\tau_1}}{\mathtt{inl}(c) \in N_{\tau_1 + \tau_2}} \qquad \frac{c \in N_{\tau_2}}{\mathtt{inr}(c) \in N_{\tau_1 + \tau_2}}$$

Recursive types:
$$\frac{c \in N_{u_i} \quad T_i = u_i \text{ is recursive}}{\mathtt{abs}(c) \in N_{T_i}}$$

Nonrecursive types:
$$\frac{c \in N_{u_i} \quad T_i = u_i \text{ is not recursive}}{c \in N_{T_i}}$$

$$\frac{c \in N_{T_i} \quad T_i = u_i \text{ is not recursive}}{c \in N_{u_i}}$$

There are no other normal forms. These rules are not enough to specify the operational semantics. We need to specify exactly how each term is reduced. Evaluation of terms needs to be

lazy (except for the `let` construct) and therefore terms must only be reduced as far as is needed for the reduction step to complete. There is a need for another kind of normal form, the so called head normal form. A term in head normal form need not be in normal form, but it has been reduced just enough to be used in a reduction step. As before, we define the head normal forms as sets per closed type

C_τ = the set of head normal forms for closed type τ

by means of the following rules

Integers:
$$\frac{n : \mathrm{Z}}{n \in C_{\mathrm{Z}}}$$

Pairs:
$$\frac{t_1 : \tau_1 \quad t_2 : \tau_2 \quad t_1 \text{ and } t_2 \text{ are closed}}{(t_1, t_2) \in C_{\tau_1 * \tau_2}}$$

Sums:
$$\frac{t : \tau_1 \quad t \text{ is closed}}{\mathtt{inl}(t) \in C_{\tau_1 + \tau_2}} \qquad \frac{t : \tau_2 \quad t \text{ is closed}}{\mathtt{inr}(t) \in C_{\tau_1 + \tau_2}}$$

Functions:
$$\frac{\lambda x.t : \tau_1 \text{ -> } \tau_2 \quad \lambda x.t \text{ is closed}}{\lambda x.t \in C_{\tau_1 \text{ -> } \tau_2}}$$

Recursive types:
$$\frac{c \in C_{u_i} \quad T_i = u_i \text{ is recursive}}{\mathtt{abs}(c) \in C_{T_i}}$$

Nonrecursive types:
$$\frac{c \in C_{u_i} \quad T_i = u_i \text{ is not recursive}}{c \in C_{T_i}}$$

$$\frac{c \in C_{T_i} \quad T_i = u_i \text{ is not recursive}}{c \in C_{u_i}}$$

We shall use $c, c_0, c_1, \ldots$ as a notation for terms in (head) normal form. From the context it will be clear if a normal form or a head normal form is indicated.

4.8 An operational semantics

Now an operational semantics of Lager can be given. A term in head normal form is not necessarily in normal form and may have subterms that can be reduced further. To ensure that the outermost redexes are reduced first, a term is reduced first to head normal form and subsequently its subterms are reduced in the same way. Two reduction relations are needed

1. $t \dashrightarrow c \iff$ term t reduces to head normal form c
2. $t \longrightarrow c \iff$ term t reduces to normal form c

The head normal form reduction relation also uses the normal form reduction relation, but only in the reduction of the `let`-statement.

The reduction relations will be introduced using rules. Some of the rules are dependent on the function definitions that are given. Rewriting a term to (head) normal form corresponds to building a proof tree.

Head normal form reduction rules

Head normal forms:
$$\frac{c \in C_\tau \quad \text{for some type } \tau}{c \dashrightarrow c}$$

Arithmetic:
$$\frac{t_1 \dashrightarrow n_1 \quad t_2 \dashrightarrow n_2}{t_1 \ \mathbf{op} \ t_2 \dashrightarrow n_1 \ op \ n_2} \quad \text{where} \quad \begin{array}{lcl} \mathbf{op} & = & +, -, * \\ op & = & \text{mathematical version of } \mathbf{op} \end{array}$$

$$\frac{t_1 \dashrightarrow n_1 \quad t_2 \dashrightarrow n_2 \quad n_2 \not\equiv 0}{t_1 \;/\; t_2 \dashrightarrow n_1 \operatorname{div} n_2}$$

$$\frac{t_1 \dashrightarrow n_1 \quad t_2 \dashrightarrow n_2 \quad n_2 \not\equiv 0}{t_1 \;\%\; t_2 \dashrightarrow n_1 \bmod n_2}$$

Relational:

$$\frac{t_1 \dashrightarrow n_1 \quad t_2 \dashrightarrow n_2 \quad n_1 \equiv n_2}{t_1 = t_2 \dashrightarrow 0}$$

$$\frac{t_1 \dashrightarrow n_1 \quad t_2 \dashrightarrow n_2 \quad n_1 \not\equiv n_2}{t_1 = t_2 \dashrightarrow 1}$$

$$\frac{t_1 \dashrightarrow n_1 \quad t_2 \dashrightarrow n_2 \quad n_1 < n_2}{t_1 < t_2 \dashrightarrow 0}$$

$$\frac{t_1 \dashrightarrow n_1 \quad t_2 \dashrightarrow n_2 \quad n_2 \leqslant n_1}{t_1 < t_2 \dashrightarrow 1}$$

Conditional:

$$\frac{t_0 \dashrightarrow n \quad n \equiv 0 \quad t_1 \dashrightarrow c}{\texttt{if } t_0 \texttt{ then } t_1 \texttt{ else } t_2 \dashrightarrow c}$$

$$\frac{t_0 \dashrightarrow n \quad n \not\equiv 0 \quad t_2 \dashrightarrow c}{\texttt{if } t_0 \texttt{ then } t_1 \texttt{ else } t_2 \dashrightarrow c}$$

Products:

$$\frac{t \dashrightarrow (t_1, t_2) \quad t_1 \dashrightarrow c}{\texttt{fst}(t) \dashrightarrow c} \qquad \frac{t \dashrightarrow (t_1, t_2) \quad t_2 \dashrightarrow c}{\texttt{snd}(t) \dashrightarrow c}$$

Sums:

$$\frac{t \dashrightarrow \texttt{inl}(t') \quad t_1[t'/x_1] \dashrightarrow c}{\texttt{case } t \texttt{ of inl}(x_1).\, t_1,\ \texttt{inr}(x_2).\, t_2 \dashrightarrow c}$$

$$\frac{t \dashrightarrow \texttt{inr}(t') \quad t_2[t'/x_2] \dashrightarrow c}{\texttt{case } t \texttt{ of inl}(x_1).\, t_1,\ \texttt{inr}(x_2).\, t_2 \dashrightarrow c}$$

Let statement:

$$\frac{t_1 \longrightarrow c_1 \quad t_2[c_1/x_1] \dashrightarrow c}{\texttt{let } x_1 \texttt{ <= } t_1 \texttt{ in } t_2 \dashrightarrow c}$$

Application:

$$\frac{t_1 \dashrightarrow \lambda x_1.t_1' \quad t_1'[t_2/x_1] \dashrightarrow c}{(t_1 \,.\, t_2) \dashrightarrow c}$$

Function definitions:

$$\frac{d_i \dashrightarrow c}{f_i \dashrightarrow c}$$

Recursive types:

$$\frac{t \dashrightarrow c}{\texttt{abs}(t) \dashrightarrow \texttt{abs}(c)} \qquad \frac{t \dashrightarrow \texttt{abs}(c)}{\texttt{rep}(t) \dashrightarrow c}$$

Normal form reduction rules

Integers:

$$\frac{t \dashrightarrow n}{t \longrightarrow n}$$

Pairs:

$$\frac{t \dashrightarrow (t_1, t_2) \quad t_1 \longrightarrow c_1 \quad t_2 \longrightarrow c_2}{t \longrightarrow (c_1, c_2)}$$

Sums:

$$\frac{t \dashrightarrow \texttt{inl}(\bullet)}{t \longrightarrow \texttt{inl}(\bullet)} \qquad \frac{t \dashrightarrow \texttt{inl}(t_1) \quad t_1 \longrightarrow c}{t \longrightarrow \texttt{inl}(c)}$$

$$\frac{t \dashrightarrow \texttt{inr}(\bullet)}{t \longrightarrow \texttt{inr}(\bullet)} \qquad \frac{t \dashrightarrow \texttt{inr}(t_1) \quad t_1 \longrightarrow c}{t \longrightarrow \texttt{inr}(c)}$$

Recursive types:

$$\frac{t \dashrightarrow \texttt{abs}(t_1) \quad t_1 \longrightarrow c}{t \longrightarrow \texttt{abs}(c)}$$

Example. The reduction to head normal form of `From.7` yields `abs(inr((7, From.(7+1))))`, but reduction to normal form never terminates because the normal form is infinite.

The operational semantics should be deterministic, i.e. the outcome should be the same everytime one reduces the same expression. This can be verified for this semantics by means of rule induction.

To make a semantics deterministic one must carefully specify the outcome of every operation. For example, consider the rules for < (or =). It has been stated that *false* could be represented by every non-zero integer. Thus the *false* outcome of < (or =) could as well be 42, or maybe more complex: a nonzero integer that may be the outcome of a function that takes one of the operands of < (=) as input. To make this operational semantics simple and deterministic we have chosen to let the *false* outcome of < and = be 1.

At this point an implementation of Lager can be made. We have made a function in both Haskell and Gofer that reduces a representation of a Lager term to a normal form, if possible, with respect to some function definitions. No type information is needed and no type checking is done, but we require that the term and function definition are well-typed for the outcome to make sense.

The version in Haskell, see appendix A, makes use of monads which allow for (some) control over the evaluation order as well as for a concise notation. For an introduction to monads in Haskell (and Haskell itself) see [Bir98].

The Gofer version, see appendix B, does not use monads. Consequently, forcing the evaluation of a term as is needed for the `let` construct, is somewhat awkward but possible. We achieved this by using case statements to somewhat force evaluation of a Gofer term.

The implementations of Lager in Haskell (monads) and Gofer (no monads), give insight in both the language Lager itself as well as insight in monads. Lager is not really ground-breaking or new because one can already influence the evaluation order in a language like Haskell with monads. But monads are more versatile than the `let` statement, they can be used for example for concurrency, see [Jon00]. Without monads we are also able to force evaluation of terms, and this might be worth further investigation.

4.9 A denotational semantics

In denotational semantics a meaning is given to a program fragment by associating it with an element in a domain of possible meanings. This domain is dependent on the type of the fragment. In this chapter we do not use domains like those introduced by Dana Scott, but information systems (see Chapter 2). To associate an information system with each closed type we need a type environment χ that assigns information systems to each type variable. Let χ be a type environment, we then define:

$$\begin{aligned}
\mathcal{V}[\![\mathbf{0}]\!]_\chi &= \mathbf{0} \\
\mathcal{V}[\![\mathbf{Z}]\!]_\chi &= \mathcal{Z} \\
\mathcal{V}[\![\tau_1 * \tau_2]\!]_\chi &= (\mathcal{V}[\![\tau_1]\!]_\chi)_\perp \times (\mathcal{V}[\![\tau_2]\!]_\chi)_\perp \\
\mathcal{V}[\![\tau_1 + \tau_2]\!]_\chi &= (\mathcal{V}[\![\tau_1]\!]_\chi)_\perp + (\mathcal{V}[\![\tau_2]\!]_\chi)_\perp \\
\mathcal{V}[\![\tau_1 \text{ -> } \tau_2]\!]_\chi &= (\mathcal{V}[\![\tau_1]\!]_\chi)_\perp \to (\mathcal{V}[\![\tau_2]\!]_\chi)_\perp \\
\mathcal{V}[\![X]\!]_\chi &= \chi(X)
\end{aligned}$$

with $X \in \mathbf{TVar}$.

After these definitions we can now assign information systems to closed types defined in the type definitions. If $T_i = u_i$ is not a recursive type definition then we take

$$\mathcal{V}[\![T_i]\!]_\chi = \mathcal{V}[\![u_i]\!]_{\chi'}$$

where

$$\chi' = \chi[\mathcal{V}[\![T_1]\!]_\chi / T_1, \ldots, \mathcal{V}[\![T_{i-1}]\!]_\chi / T_{i-1}]$$

for an arbitrary type environment χ.

If $T_i = u_i$ is a recursive type definition then we take

$$\mathcal{V}[\![T_i]\!]_\chi = \text{the } \trianglelefteq\text{-least fixed point of } I \longmapsto \mathcal{V}[\![u_i]\!]_{\chi'}$$

where

$$\chi' = \chi[\mathcal{V}[\![T_1]\!]_\chi / T_1, \ldots, \mathcal{V}[\![T_{i-1}]\!]_\chi / T_{i-1}, I / T_i]$$

for an arbitrary type environment χ.

Note that the type environment can be taken arbitrarily since we only assign information systems to closed types.

Had we relaxed the restrictions on the type definitions and allowed mutually recursive type definitions, then a more complex, simultaneous fixpoint construction would have been necessary. To keep matters relatively simple we have chosen the easy way out.

We now have a way to associate a closed type τ with an information system of values for that type. We write

$$\mathcal{V}_\tau = \mathcal{V}[\![\tau]\!]_\chi$$

for an arbitrary type environment χ.
This information system $\mathcal{V}_\tau$ has an underlying cpo of elements of the information systems, $|\mathcal{V}_\tau|$. Computation of term of a closed type τ may either diverge or return a result, so the information system (and thus the underlying cpo) is lifted.

To evaluate a term means to rewrite it under the given function definitions. A function is usually called with some arguments which will be substituted for the variables bound by abstraction. A variable environment is needed that captures this substitution of arguments for variables when a function is applied. A variable environment is a function ρ that assigns a denotation to each variable, i.e.

$$\rho : \mathbf{Var} \to \bigcup\{|\mathcal{V}_{\tau_\perp}| \mid \tau \text{ is a closed type}\}$$

We write **VEnv** for the cpo of all variable environments, i.e.

$$\mathbf{VEnv} = [\,\mathbf{Var} \to \bigcup\{|\mathcal{V}_{\tau_\perp}| \mid \tau \text{ is a closed type}\}\,]$$

A function itself is defined by a term and thus has a denotation. We can view the system of function definitions as a function that given a function variable returns the denotation of that function. A function environment is a function φ that assigns a denotation to each function variable, i.e.

$$\varphi : \mathbf{FVar} \to \bigcup\{|\mathcal{V}_{\tau_\perp}| \mid \tau \text{ is a closed type}\}$$

We write **FEnv** for the cpo of all function variable environments, i.e.

$$\mathbf{FEnv} = [\,\mathbf{FVar} \to \bigcup\{|\mathcal{V}_{\tau_\perp}| \mid \tau \text{ is a closed type}\}\,]$$

A term t of closed type τ will now be evaluated in a function environment φ and variable environment ρ, and denote an element in the cpo $|\mathcal{V}_{\tau_\perp}|$, i.e.

$$[\![t]\!]\varphi\rho \in |\mathcal{V}_{\tau_\perp}|$$

or

$$[\![t]\!] \in [\mathbf{FEnv} \to [\mathbf{VEnv} \to |\mathcal{V}_{\tau\perp}|]]$$

The function environment is dependent on the function definitions. We need the correct function environment before we can calculate denotations of terms given the function definitions. **FEnv** is a cpo and so the cpo of all continuous functions from **FEnv** to **FEnv** is also a cpo, and this cpo has a fixpoint constructor. We will now construct the function environment as being the least fixpoint of the continuous function $F : \mathbf{FEnv} \to \mathbf{FEnv}$ given by:

$$F(\phi) = \phi[[\![t_1]\!]\phi\rho \,/\, f_1, \ldots, [\![t_k]\!]\phi\rho \,/\, f_k]$$

for an arbitrary variable environment ρ. Indeed, ρ can be arbitrary since each of the defining terms $t_1, \ldots, t_k$ is closed.

It is easy to verify that function F is continuous. We can therefore choose its least fixpoint as our function environment.

$$\varphi = \mathrm{fix}(F) = \bigsqcup_n F^n(\perp)$$

Next we define the denotations of terms as

$$\begin{aligned}
[\![\bullet]\!]\varphi\rho &= \perp \\
[\![x]\!]\varphi\rho &= \rho(x) \\
[\![n]\!]\varphi\rho &= \lfloor n \rfloor \\
[\![t_1 \texttt{ + } t_2]\!]\varphi\rho &= \mathit{let}\ n_1 \Leftarrow [\![t_1]\!]\varphi\rho, n_2 \Leftarrow [\![t_2]\!]\varphi\rho\ .\ \lfloor n_1 + n_2 \rfloor \\
[\![t_1 \texttt{ - } t_2]\!]\varphi\rho &= \mathit{let}\ n_1 \Leftarrow [\![t_1]\!]\varphi\rho, n_2 \Leftarrow [\![t_2]\!]\varphi\rho\ .\ \lfloor n_1 - n_2 \rfloor \\
[\![t_1 \texttt{ * } t_2]\!]\varphi\rho &= \mathit{let}\ n_1 \Leftarrow [\![t_1]\!]\varphi\rho, n_2 \Leftarrow [\![t_2]\!]\varphi\rho\ .\ \lfloor n_1 \times n_2 \rfloor \\
[\![t_1 \texttt{ / } t_2]\!]\varphi\rho &= \mathit{let}\ n_1 \Leftarrow [\![t_1]\!]\varphi\rho, n_2 \Leftarrow [\![t_2]\!]\varphi\rho\ .\ \mathit{cond}(\mathit{iszero}(n_2), \perp, n_1\ \mathit{div}\ n_2) \\
[\![t_1 \texttt{ \% } t_2]\!]\varphi\rho &= \mathit{let}\ n_1 \Leftarrow [\![t_1]\!]\varphi\rho, n_2 \Leftarrow [\![t_2]\!]\varphi\rho\ .\ \mathit{cond}(\mathit{iszero}(n_2), \perp, n_1\ \mathit{mod}\ n_2) \\
[\![t_1 \texttt{ < } t_2]\!]\varphi\rho &= \mathit{let}\ n_1 \Leftarrow [\![t_1]\!]\varphi\rho, n_2 \Leftarrow [\![t_2]\!]\varphi\rho\ .\ \mathit{less}(n_1, n_2) \\
&\quad \text{where}\ \ \mathit{less}(n_1, n_2) = \lfloor 0 \rfloor, \text{ if } n_1 < n_2 \\
&\qquad\qquad \mathit{less}(n_1, n_2) = \lfloor 1 \rfloor, \text{ if } n_2 \leq n_1 \\
[\![t_1 \texttt{ = } t_2]\!]\varphi\rho &= \mathit{let}\ n_1 \Leftarrow [\![t_1]\!]\varphi\rho, n_2 \Leftarrow [\![t_2]\!]\varphi\rho\ .\ \mathit{equal}(n_1, n_2) \\
&\quad \text{where}\ \ \mathit{equal}(n_1, n_2) = \lfloor 0 \rfloor, \text{ if } n_1 = n_2 \\
&\qquad\qquad \mathit{equal}(n_1, n_2) = \lfloor 1 \rfloor, \text{ if } n_1 \neq n_2 \\
[\![\texttt{if } t_0 \texttt{ then } t_1 \texttt{ else } t_2]\!]\varphi\rho &= \mathit{Cond}([\![t_0]\!]\varphi\rho, [\![t_1]\!]\varphi\rho, [\![t_2]\!]\varphi\rho) \\
[\![(t_1 \texttt{ , } t_2)]\!]\varphi\rho &= \lfloor ([\![t_1]\!]\varphi\rho, [\![t_2]\!]\varphi\rho) \rfloor \\
[\![\texttt{fst}(t)]\!]\varphi\rho &= \mathit{let}\ v \Leftarrow [\![t]\!]\varphi\rho\ .\ \pi_1(v) \\
[\![\texttt{snd}(t)]\!]\varphi\rho &= \mathit{let}\ v \Leftarrow [\![t]\!]\varphi\rho\ .\ \pi_2(v) \\
[\![\texttt{inl}(t)]\!]\varphi\rho &= \lfloor \mathit{in}_1([\![t]\!]\varphi\rho) \rfloor \\
[\![\texttt{inr}(t)]\!]\varphi\rho &= \lfloor \mathit{in}_2([\![t]\!]\varphi\rho) \rfloor
\end{aligned}$$

$$[\![\texttt{case } t \texttt{ of inl}(x_1).t_1,\ \texttt{inr}(x_2).t_2]\!]\varphi\rho = \textit{let } v \Leftarrow [\![t]\!]\varphi\rho \,.\, \textit{case } [\![t]\!]\varphi\rho \textit{ of } in_1(d_1).[\![t_1]\!]\varphi\rho[d_1/x_1] \mid in_2(d_2).[\![t_2]\!]\varphi\rho[d_2/x_2].$$

$$[\![\texttt{let } x_1 \texttt{ <= } t_1 \texttt{ in } t_2]\!]\varphi\rho = \textit{let } v \Leftarrow [\![t_1]\!]\varphi\rho \,.\, [\![t_2]\!]\varphi\rho[v/x]$$

$$[\![(t_1.t_2)]\!]\varphi\rho = \textit{let } r \Leftarrow [\![t_1]\!]\varphi\rho \,.\, r([\![t_2]\!]\varphi\rho)$$

$$[\![\lambda x.t]\!]\varphi\rho = \lfloor \lambda d \in |\mathcal{V}_{type(x)_\perp}| \,.\, [\![t]\!]\varphi\rho[d/x] \rfloor$$

$$[\![f]\!]\varphi\rho = \varphi(f)$$

$$[\![\texttt{abs}(t)]\!]\varphi\rho = [\![t]\!]\varphi\rho$$

$$[\![\texttt{rep}(t)]\!]\varphi\rho = [\![t]\!]\varphi\rho$$

4.10 Remarks on Lager

Lager itself is quite low level and should not be seen as a real language, but rather a language in which other functional languages might be translated. That is, we can study other functional languages by trying to define similar constructions in Lager.

For example, defining the type `Zlist` of lists of integers in the Lager type system is quite awkward compared to the way this is done in Haskell

```
data List = Nil | Cons Int List
```

This constructor could be translated into Lager as a type definition with function declarations and definitions for the *constructors* `Nil` and `Cons`.

```
List = 0 + (Z * List)
Nil : Zlist
Cons : Z -> (List -> List)
z : Z
l : List
Nil = abs(inl(@))
Cons = \z . ( \l . abs(inr( (z,l) )) )
```

Some instructions like `case` will need to be adapted to allow discrimination of the different constructors, that is we want a case construction like

`case l of Nil.`t_1`,Cons(z,l').`t_2

This idea can be generalized to more complex definitions. The `case` construction must then be adapted to be able to choose between not just two alternatives but between as many constructors as there are in the definition.

In the ideal case, operational and denotational semantics agree on the convergence of a term. When that is the case, we speak of the *adequacy* of the denotational with respect to the operational semantics and vice versa. In the operational semantics a well-typed, closed term converges if it can be rewritten to a normal form. In the denotational semantics such a term converges if its denotation differs from $\perp$. We have not made any claims on the agreement of both operational and denotational semantics. Showing this would require a tedious long and complex proof and that was not the objective of this paper.

A closely related problem is that of full abstractness of a semantic model. If each object in the model is a valid / realizable program we call the model fully abstract. More on full abstractness of semantic models can be found in chapter 11 section 10 of [Win92] and the section "PCF and the Problem of Full Abstraction" of [Jun96]. This last article also gives an overview of the developments in denotational semantics.

Chapter 5

Proof methods

The denotational semantics provides us with some proof principles we can use to prove properties of programs or functions.

5.1 Fixpoint induction and partial object induction

The properties we can prove with the fixpoint induction due to Dana Scott are chain complete properties.

Definition. Let D be a cpo. A predicate P on elements of the cpo D is *chain complete* iff all ω-chains $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$ in D satisfy

$$(\forall\, n \in \omega :: P(d_n)) \quad \Rightarrow \quad P(\bigsqcup_{n\in\omega} d_n)$$

Remark. In the literature[1] such a predicate or property is also called *admissable* or *inclusive*. Some texts on domain theory call cpos "chain complete posets".

There are methods for constructing chain complete predicates. Consult [Win92] or any good text on domain theory for the constructions. These methods generate only a subset of all chain complete predicates. Checking self constructed properties for chain completeness is tedious.

Theorem. (*Fixpoint induction*)
Let D be a cpo with bottom and let $f : D \to D$ be a continuous function. Let P be a chain complete predicate on D. If

1. $P(\bot)$, and
2. $(\forall\, d \in D :: P(d) \Rightarrow P(F(d)))$

then

$P(\mathrm{fix}(F))$

Proof. Recall that $\mathrm{fix}(F) = \bigsqcup_n F^n(\bot)$. Requirement 1 gives that P holds for $F^0(\bot) = \bot$. Requirement 2 combined with previous observation yields by natural induction that $(\forall\, n \in \omega :: P(F^n(\bot)))$. Combined with the chain completeness of P this implies that $P(\bigsqcup_n F^n(\bot)) = P(\mathrm{fix}(F))$. □

The denotations of functions in a functional language like Lager are generally taken to be the least fixpoint of some function F as we have seen in section 4.9. With fixpoint induction we can prove (chain complete) properties of a fixpoint and thus of the function it denotes.

Not all properties that hold for the fixpoint can be stated as chain complete predicates. So we cannot use fixpoint induction to prove *all* properties of a fixpoint, i.e. fixpoint induction is not complete.

[1] In accordance to the theory the terminology is also quite rich

Functions act on data of some type. A type has an information system which is the least fixpoint to the type equation used to define the type. The elements of the (data) type may be infinite. When a type has infinite elements, the cpo for the type can not be inductively defined as this would only generate cpos with finite elements. This was one of the reasons to resort to information systems, because they only work with finite elements and use a completion proces to generate the infinite elements. However, this still indicates a problem concerning the use of induction on the structure of data types for proving properties. By structural induction, that is induction on the way the type is built up, one only proves that the property holds for (all) finite elements. Or using fixpoint induction one only proves that the chain complete property holds for the limits of finite elements. If all infinite elements are limits of finite elements then the property holds indeed for all elements, infinite or finite, of the type. To make this formal we need some definitions.

Definition. Let D be a cpo. An element $e \in D$ is *finite* iff for each ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$ in D such that $e \sqsubseteq \bigsqcup_n d_n$, there exists an $i \in \omega$ for which $e \sqsubseteq d_i$. The set of all finite elements of D will be denoted by D^0.

Definition. Let D be a cpo. D is called *ω-algebraic* iff D^0 is countable and for each element (finite or infinite) $d \in D$ there exists an ω-chain of finite elements $e_0 \sqsubseteq e_1 \sqsubseteq \cdots \sqsubseteq e_n \sqsubseteq \cdots$ in D such that $d = \bigsqcup_n e_n$.

Remark. An other name in domain theory for ω-algebraic is "chain inductive".

The set of finite elements must be countable, because induction relies on a well-ordering of the elements and an uncountable set clearly has no well ordering. The constructions on the information systems need to preserve the ω-algebraic property of the underlying cpos. This shown for the information systems we use, in chapter 12 of [Win92].

The ω-algebraic property restricts the set of finite elements to be countable, but the set of finite elements should also be well-founded. All constructions on information systems with exception of the lifted function space preserve well-foundedness of the set of finite elements of the underlying cpo. That is, if the set of finite elements of a cpo D is well-founded, then the set of finite elements of the lifted cpo $D_\perp$ is also well-founded. Similar facts hold for sum and product.

This limits the applicability of the following theorem to those datatypes that can be (recursively) built from admissable cpos, i.e. ω-algebraic cpos whose finite elements form well-founded sets, using only lifting, sum and product.

Theorem. (*partial object induction*)
Let D be a *ω-algebraic* cpo whose finite elements form a well-founded set. Let P be a chain complete predicate on D. Then P holds for *all* elements of D if for all finite elements $e \in D$, $P(e)$ holds on the assumption that $P(e')$ holds whenever $e' \sqsubset e$. That is,

$$(\forall\, e \in D : \text{e is finite} : (\forall\, e' \in D : e' \sqsubset e : P(e') \Rightarrow P(e))) \quad \Rightarrow \quad (\forall\, d \in D :: P(d))$$

Proof. The set of all finite elements of D is a well-founded set, so the first part of the implication implies (by structural induction) that P holds for all finite elements. For each ω-chain of finite elements $e_0 \sqsubseteq e_1 \sqsubseteq \cdots \sqsubseteq e_n \sqsubseteq \cdots$ we have that $P(\bigsqcup_n e_n)$ holds because P is chain complete. All elements $d \in D$, finite or infinite, are the limit of a ω-chain of finite elements, because D is ω-algebraic. Hence $P(d)$ holds for all $d \in D$.

Remark. A similar but incorrect theorem appears in [Hen87].

With this result we can conclude that the proof rules in chapter 9 of [Bir98] for proving a property P for all finite and infinite lists are valid. But we must ensure that in the semantics the cpos for the data types are ω-algebraic and the sets of finite elements are well-founded. We have already remarked that the cpos for the data types in Lager are ω-algebraic and that lifting, sum and product preserve well-foundedness of the set of finite elements. We made a comment on translating type constructions from Haskell to Lager, so it would seem that the types in Haskell indeed have ω-algebraic cpos. Lists can be built up using lifting, sum and product and thus the induction principle in [Bir98] is sound.

5.2 Approximation

Partial object induction is applicable when there is an explicit list (or other data structure) in the program to induct upon. But sometimes the list is implicit. Chapter 9 of [Bir98] discusses this with the *map-iterate property* and presents approximation as a solution. With the approximation lemma a chain of finite approximations to the data structure is got. With natural induction the property is shown to hold for all those approximations and by virtue of the argument in the previous section, the property holds for the infinite case.

The approximation function for lists is defined in Haskell as

```
approx (n+1) []     = []
approx (n+1) (x:xs) = x : approx n xs
```

Because there is no rule when the first argument is zero, we have

`approx 0 xs` = $\bot$ for all lists xs

This function is used to get a chain of finite elements

$$\texttt{approx 0 xs} \sqsubseteq \texttt{approx 1 xs} \sqsubseteq \cdots \sqsubseteq \texttt{approx n xs} \sqsubseteq \cdots$$

The *approximation lemma* ([GH99]) is now given as

$$(\forall \text{ lists } \texttt{xs, ys} :: \texttt{xs = ys} \iff (\forall \texttt{n} :: \texttt{approx n xs = approx n ys}))$$

Example. Suppose we define the following functions

```
ones = 1 : ones
twos = 2 : twos
```

and we suppose we want to prove

```
map (+1) ones = twos
```

We prove this by proving the following by induction.

$$P(\texttt{n}) \equiv \texttt{approx n (map (+1) ones)} = \texttt{approx n twos})$$

Proof.

Base case. $P(0)$ holds trivially because both sides equal $\bot$. Although we have $\bot \neq \bot$ computationally, mathematically we have $\bot = \bot$.

Induction step. Assume that $P(\texttt{n})$ holds for a natural number n. Then

```
    approx (n + 1) (map (+1) ones)
= { definition of ones }
    approx (n + 1) (map (+1) (1 : ones))
= { definition of map, calculus }
    approx (n + 1) (2 : (map (+1) ones))
= { definition of approx }
    2 :  (approx n (map (+1) ones))
= { induction hypothesis }
    2 :  (approx n twos)
= { definition of approx }
    approx (n + 1) (2 : twos)
= { definition of twos}
    approx (n + 1) twos
```

Thus $P(\texttt{n+1})$ holds whenever $P(\texttt{n})$ holds for a natural number n, and thus $P(\texttt{n})$ holds for all natural numbers n and by the approximation lemma we have proved that

```
map (+1) ones = twos
```

□

5.3 Other methods

Approximation is not the only applicable approach. The article [GH99] shows the use of three other methods.

The first method is fixpoint induction which was discussed in the previous section. Fixpoint induction is sound but, as remarked before, not complete.

A second method is coinduction which is closely related to the operational semantics. Coinduction is sound and complete.

The third method is fusion. In this method the functions are expressed in terms of an operator `unfold` which has a so called universal property. Proofs rely on the use of this universal property. This method is not connected to a denotational semantics of the language, contrary to fixpoint induction and approximation. Fusion is also discussed in [Bir98] in the context of deriving functions from their specifications. This is, of course, closely related to proving properties.

Chapter 6

Conclusions and further work

The denotational semantics of (typed) Lapa cannot be given, without some major repairments. This led us to define our own language, Lager, for which a denotational semantics could be given. After defining a denotational semantics we studied some proof methods and concluded the soundness of the proof principle from chapter 9 of [Bir98], which we have called here partial object induction. This principle is sound on the occasion that the cpos for the data types are ω-algebraic. A denotational semantics for the language is needed to prove this induction principle. There are other proof methods which are not dependent on the existence of a denotational semantics, such as coinduction and fusion. More information on both principles with a working example can be found in [GH99].

6.1 Further work

We have not shown or claimed the adequacy and/or the full abstractness of the denotational semantics, and we have left this as further work.

Lager is very low-level and at the end of chapter 4 we remarked that the data type construction from like Haskell might be translated to Lager. Another construction is pattern matching. It might be interesting to check for example how the approximation function defined by pattern matching can be translated into a Lager definition. More aspects of higher level languages might be studied by translating them, or by defining a denotation for them.

The constructions `rep` and `abs` serve to unfold and fold a recursive type. These constructions might be eliminated and we leave it as further work to check how this can be done.

In the implementation of Lager in a lazy functional language we needed the ability to force evaluation of an expression before using it further. Haskell has monads which can be used to force evaluation. Gofer has no monads, but we still found a way to force the evaluation which might be studied in more detail.

The implementations of Lager were based on the operational semantics. An implementation based on the denotational semantics should also be possible.

Appendix A

An implementation of Lager in Haskell98

```
data Tree = Diverge                                  -- divergerende term
          | Num Int                                  -- number
          | Var String                               -- variabele
          | FVar String                              -- function variabele
          | Pair Tree Tree                           -- pairs
          | UnOp String Tree                         -- unary operators
          | BinOp Char Tree Tree                     -- binary operators
          | Cond Tree Tree Tree                      -- conditional
          | CaseStmt Tree String Tree String Tree -- case
          | Lam String Tree                          -- abstraction
          | Appl Tree Tree                           -- application
          | LetStmt String Tree Tree                 -- let


instance Show Tree where
  show Diverge  = "@"
  show (Num n)  = show n
  show (Var x)  = x
  show (FVar f) = f
  show (Pair t1 t2) = "(" ++ show t1 ++ "," ++ show t2 ++ ")"
  show (UnOp op t1) = op ++ "(" ++ show t1 ++ ")"
  show (BinOp op t1 t2) = "(" ++ show t1 ++ [op] ++ show t2 ++ ")"
  show (Cond t0 t1 t2) =
    "(IF " ++ show t0 ++ " THEN " ++ show t1 ++ " ELSE " ++ show t2 ++ ")"
  show (CaseStmt t0 x1 t1 x2 t2) =
    "(CASE " ++ show t0 ++ " OF INL(" ++ x1 ++ ")." ++ show t1 ++
                            ", INR(" ++ x2 ++ ")." ++ show t2 ++ ")"
  show (Lam x1 t1) = "(\\" ++ x1 ++ "." ++ show t1 ++ ")"
  show (Appl t1 t2) = "(" ++ show t1 ++ "." ++ show t2 ++ ")"
  show (LetStmt x1 t1 t2) =
    "(LET " ++ x1 ++ "<-" ++ show t1 ++ " IN " ++ show t2 ++ ")"


-- subst v s t =
-- subtitute a term s for a free variable v in term t
-- or subst v s t = t[s/v]

subst :: String -> Tree -> Tree -> Tree
subst var s vx@(Var x) =
  if var == x then s else vx
```

```
subst var s (Pair t1 t2) =
  Pair (subst var s t1) (subst var s t2)
subst var s (UnOp op t1) =
  UnOp op (subst var s t1)
subst var s (BinOp c t1 t2) =
  BinOp c (subst var s t1) (subst var s t2)
subst var s (Cond t0 t1 t2) =
  Cond (subst var s t0) (subst var s t1) (subst var s t2)
subst var s (Appl t1 t2) =
  Appl (subst var s t1) (subst var s t2)
subst var s lk@(Lam x t1) =
  if var == x then lk else Lam x (subst var s t1)
subst var s (CaseStmt t0 x1 t1 x2 t2) =
  CaseStmt (subst var s t0) x1 u1 x2 u2
  where -- x1 is only bound in term t1 and x2 is only bound in term t2
    u1 = if var == x1 then t1 else (subst var s t1)
    u2 = if var == x2 then t2 else (subst var s t2)
subst var s (LetStmt x1 t1 t2) =
  LetStmt x1 (subst var s t1) (if var == x1 then t2 else (subst var s t2))
subst var s c = c  -- c is @ or a number or a function variable


doAritm :: Char -> Int -> Int -> Maybe Tree
doAritm op n1 n2 =
  case op of
    '+' -> return (Num (n1 + n2))
    '-' -> return (Num (n1 - n2))
    '*' -> return (Num (n1 * n2))
    '/' -> if n2 == 0 then Nothing else return (Num (n1 `div` n2))
    '%' -> if n2 == 0 then Nothing else return (Num (n1 `mod` n2))
    '<' -> if n1  < n2 then return (Num 0) else return (Num 1)
    '=' -> if n1 == n2 then return (Num 0) else return (Num 1)


-- reduce mem t = try to reduce t to a head form under the
-- the definitions in mem.
-- we use the primitive lookup :: Eq a => a -> [(a, b)] -> Maybe b
-- which is defined in the standard prelude.

reduce :: [(String, Tree)] -> Tree -> Maybe Tree

reduce mem Diverge = Nothing -- no head normal form

reduce mem (FVar f) =
  do d <- lookup f mem
     reduce mem d

reduce mem u@(UnOp op t1)
 | op == "INL" || op == "INR" = return u
 | op == "ABS"                = do c1 <- reduce mem t1
                                   return (UnOp op c1)
 | op == "REP"                = do (UnOp "ABS" c1) <- reduce mem t1
                                   return c1
 | op == "FST"                = do (Pair t11 t12)  <- reduce mem t1
                                   reduce mem t11
 | op == "SND"                = do (Pair t11 t12)  <- reduce mem t1
                                   reduce mem t12
 | otherwise                  = Nothing
```

```
reduce mem (BinOp op t1 t2) =
  do (Num n1) <- reduce mem t1
     (Num n2) <- reduce mem t2
     doAritm op n1 n2


reduce mem (Cond t0 t1 t2) =
  do (Num n) <- reduce mem t0
     if n == 0
       then (reduce mem t1)
       else (reduce mem t2)


reduce mem (CaseStmt t0 x1 t1 x2 t2) =
  do (UnOp op u0) <- reduce mem t0
     case op of
       "INL" -> reduce mem (subst x1 u0 t1)
       "INR" -> reduce mem (subst x2 u0 t2)
       _     -> Nothing

reduce mem (Appl t1 t2) =
  do (Lam x0 t0) <- reduce mem t1
     reduce mem (subst x0 t2 t0)

reduce mem (LetStmt x1 t1 t2) =
  do c1 <- compute mem t1 -- first reduce t1 to a normal form
     reduce mem (subst x1 c1 t2)

reduce mem t = return t -- head normal forms: numbers, pairs, abstractions.


-- compute mem t = try to reduce t to a normal form
-- under the definitions in mem

compute :: [(String, Tree)] -> Tree -> Maybe Tree
compute mem t =
  do hnf <- reduce mem t
     case hnf of
       (Num n)      -> return (Num n)
       (Pair t1 t2) -> do c1 <- compute mem t1
                          c2 <- compute mem t2
                          return (Pair c1 c2)
       -- left and right sums and ABS
       (UnOp op Diverge) -> if op == "ABS"
                            then Nothing
                            else return (UnOp op Diverge) -- normal forms INL(©) and INR(©)
       (UnOp op t)       -> do c <- compute mem t
                               return (UnOp op c)
       -- lambda abstractions in head normal form
       -- are not in normal form. (and will never be)
       _                 -> Nothing
```

```
-- an example: some function definitions

fdefs = [ ("From",
          (Lam "n" (UnOp "ABS"
                   (UnOp "INR" (Pair (Var "n")
                                     (Appl (FVar "From")
                                           (BinOp '+' (Var "n")
                                                      (Num 1))))))),

          ("Take",
          (Lam "n"
          (Lam "xs" (Cond (Var "n")
                          (UnOp "ABS" (UnOp "INL" Diverge))
                          (CaseStmt (UnOp "REP" (Var "xs"))
                            "l" (UnOp "ABS" (UnOp "INL" Diverge))
                            "r" (UnOp "ABS" (UnOp "INR"
                                   (Pair (UnOp "FST" (Var "r"))
                                         (Appl (Appl (FVar "Take")
                                                     (BinOp '-' (Var "n")
                                                                (Num 1)))
                                               (UnOp "SND" (Var "r"))))))))))),

          ("Sum",
          (Lam "xs" (CaseStmt (UnOp "REP" (Var "xs"))
                      "l" (Num 0)
                      "r" (BinOp '+' (UnOp "FST" (Var "r"))
                                     (Appl (FVar "Sum")
                                           (UnOp "SND" (Var "r"))))))),

-- (Take.2).(From.7)

          ("TakeTwoFromSeven",
          (Appl (Appl (FVar "Take") (Num 2))
                (Appl (FVar "From") (Num 7)))),


          ("Filter",
          (Lam "b"
          (Lam "xs" (CaseStmt (UnOp "REP" (Var "xs"))
                      "l" (UnOp "ABS" (UnOp "INL" (Var "l")))
                      "r" (Cond (Appl (Var "b")
                                      (UnOp "FST" (Var "r")))
                                (UnOp "ABS" (UnOp "INR"
                                  (Pair (UnOp "FST" (Var "r"))
                                   (Appl (Appl (FVar "Filter")
                                               (Var "b"))
                                         (UnOp "SND" (Var "r"))))))
                                (Appl (Appl (FVar "Filter")
                                            (Var "b"))
                                      (UnOp "SND" (Var "r")))))))),

          ("NonDivi",
          (Lam "n"
          (Lam "m" (BinOp '<' (Num 0)
                              (BinOp '%' (Var "m")
                                         (Var "n")))))),
```

```
          ("Sieve",
          (Lam "xs" (CaseStmt (UnOp "REP" (Var "xs"))
                    "l" (UnOp "ABS" (UnOp "INL" (Var "l")))
                    "r" (UnOp "ABS" (UnOp "INR"
                          (Pair (UnOp "FST" (Var "r"))
                                (Appl (FVar "Sieve")
                                      (Appl (Appl (FVar "Filter")
                                                  (Appl (FVar "NonDivi")
                                                        (UnOp "FST" (Var "r"))))
                                            (UnOp "SND" (Var "r")))))))))),


          ("Primes",
          (Appl (FVar "Sieve") (Appl (FVar "From") (Num 2)))),

-- (Take.5).Primes

          ("FirstFivePrimes",
          (Appl (Appl (FVar "Take") (Num 5)) (FVar "Primes"))),

-- Some alternative definitions

          ("FromE",
          (Lam "n" (LetStmt "e" (Appl (FVar "FromE")
                                      (BinOp '+' (Var "n")
                                                 (Num 1)))
                                (UnOp "ABS" (UnOp "INR"
                                  (Pair (Var "n") (Var "e")))))),

          ("ATake",
          (Lam "n"
          (Lam "xs" (UnOp "ABS"
                    (Cond (Var "n")
                          (UnOp "INL" Diverge)
                          (CaseStmt (UnOp "REP" (Var "xs"))
                            "l"  (UnOp "INL" Diverge)
                            "l"  (UnOp "INR"
                                   (Pair (UnOp "FST" (Var "r"))
                                         (Appl (Appl (FVar "ATake")
                                                     (BinOp '-' (Var "n")
                                                                (Num 1)))
                                               (UnOp "SND" (Var "r"))))))))))),

          ("TakeTwoFromSevenE",
          (Appl (Appl (FVar "Take") (Num 2))
                (Appl (FVar "FromE") (Num 7))))


        ]


fiveprimeslist = compute fdefs  (FVar "FirstFivePrimes")
-- output:
-- Just ABS(INR((2,ABS(INR((3,ABS(INR((5,ABS(INR((7,ABS(INR((11,ABS(INL(¢))))))))))))))))

-- which are a lot of parentheses and and the first five primes
```

Appendix B

An implementation of Lager in Gofer

```
data Maybe a = Nothing | Just a

data Tree = Diverge                                -- diverging term
          | Num Int                                -- number
          | Var String                             -- variabele
          | FVar String                            -- function variabele
          | Pair Tree Tree                         -- pairs
          | UnOp String Tree                       -- unairy operators
          | BinOp Char Tree Tree                   -- binairy operators
          | Cond Tree Tree Tree                    -- conditional
          | CaseStmt Tree String Tree String Tree -- case
          | Lam String Tree                        -- abstraction
          | Appl Tree Tree                         -- application
          | LetStmt String Tree Tree               -- let

instance Text Tree where
  showsPrec i Diverge  = showChar '@'
  showsPrec i (Num n)  = shows n
  showsPrec i (Var x)  = showString x
  showsPrec i (FVar f) = showString f
  showsPrec i (Pair t1 t2) =
    showChar '(' . shows t1 . showChar ',' . shows t2 . showChar ')'
  showsPrec i (UnOp op t) =
    showString op . showChar '(' . shows t . showChar ')'
  showsPrec i (BinOp op t1 t2) =
    showChar '(' . shows t1 . showChar op . shows t2 . showChar ')'
  showsPrec i (Cond t0 t1 t2) =
    showString "(IF " . shows t0 . showString " THEN " . shows t1 .
    showString " ELSE " . shows t2 . showChar ')'
  showsPrec i (CaseStmt t0 x1 t1 x2 t2) =
    showString "(CASE " . shows t0 .
      showString ( " OF INL(" ++ x1 ++ ")." ) . shows t1 .
      showString (   ", INR(" ++ x2 ++ ")." ) . shows t2 . showChar ')'
  showsPrec i (Lam x1 t1) =
    showString ( "(\\" ++ x1 ++ ".") . shows t1 . showChar ')'
  showsPrec i (Appl t1 t2) =
    showChar '(' . shows t1 . showChar '.' . shows t2 . showChar ')'
  showsPrec i (LetStmt x1 t1 t2) =
    showString ( "LET " ++ x1 ++ "<-" ) . shows t1 .
      showString " IN " . shows t2 . showChar ')'
```

```
instance Text a => Text (Maybe a) where
  showsPrec i Nothing  = showString "Nothing"
  showsPrec i (Just x) = showString "Just " . shows x


-- subst v s t =
-- subtitute a term s for a free variable v in term t
-- or subst v s t = t[s/v]

subst :: String -> Tree -> Tree -> Tree
subst var s vx@(Var x) =
  if var == x then s else vx
subst var s (Pair t1 t2) =
  Pair (subst var s t1) (subst var s t2)
subst var s (UnOp op t1) =
  UnOp op (subst var s t1)
subst var s (BinOp c t1 t2) =
  BinOp c (subst var s t1) (subst var s t2)
subst var s (Cond t0 t1 t2) =
  Cond (subst var s t0) (subst var s t1) (subst var s t2)
subst var s (Appl t1 t2) =
  Appl (subst var s t1) (subst var s t2)
subst var s lk@(Lam x t1) =
  if var == x then lk else Lam x (subst var s t1)
subst var s cs@(CaseStmt t0 x1 t1 x2 t2) =
  CaseStmt (subst var s t0) x1 u1 x2 u2
  where
    u1 = if var == x1 then t1 else (subst var s t1)
    u2 = if var == x2 then t2 else (subst var s t2)
subst var s lt@(LetStmt x1 t1 t2) =
  LetStmt x1 (subst var s t1) (if var == x1 then t2 else (subst var s t2))
subst var s c = c -- c is of @, or a number or a function variable


lookup :: [(String, a)] -> String -> Maybe a
lookup table index = foldr dofind Nothing table
  where
    dofind (key, item) rest =
      if key == index then Just item else rest


doAritm :: Char -> Int -> Int -> Maybe Tree
doAritm op n1 n2 =
  case op of
    '+' -> Just (Num (n1 + n2))
    '-' -> Just (Num (n1 - n2))
    '*' -> Just (Num (n1 * n2))
    '/' -> if n2 == 0 then Nothing else Just (Num (n1 `div` n2))
    '%' -> if n2 == 0 then Nothing else Just (Num (n1 `mod` n2))
    '<' -> if n1  < n2 then Just (Num 0) else Just (Num 1)
    '=' -> if n1 == n2 then Just (Num 0) else Just (Num 1)


-- reduce mem t = try to reducce t to a head normal form under the
-- definitions in mem.

reduce :: [(String, Tree)] -> Tree -> Maybe Tree

reduce mem Diverge = Nothing

reduce mem (FVar f) =
  case (lookup mem f) of
    Nothing  -> Nothing
    (Just d) -> reduce mem d
```

```
reduce mem u@(UnOp op t1)
 | op == "INL" || op == "INR" = Just u
 | op == "ABS" = case (reduce mem t1) of
                   Nothing -> Nothing
                   (Just c1) -> Just (UnOp op c1)
 | op == "REP" = case (reduce mem t1) of
                   Nothing -> Nothing
                   (Just (UnOp "ABS" c1)) -> Just c1
                   _ -> Nothing
 | op == "FST" = case (reduce mem t1) of
                   Nothing -> Nothing
                   (Just (Pair t11 t12)) -> reduce mem t11
                   _ -> Nothing
 | op == "SND" = case (reduce mem t1) of
                   Nothing -> Nothing
                   (Just (Pair t11 t12)) -> reduce mem t12
                   _ -> Nothing
 | otherwise   = Nothing

reduce mem (BinOp op t1 t2) =
  case (reduce mem t1) of
    Nothing -> Nothing
    Just (Num n1) ->
      case (reduce mem t2) of
        Nothing -> Nothing
        Just (Num n2) -> doAritm op n1 n2
        _ -> Nothing
    _ -> Nothing

reduce mem (Cond t0 t1 t2) =
  case (reduce mem t0) of
    Nothing -> Nothing
    Just (Num n) ->
      if n == 0
        then reduce mem t1
        else reduce mem t2
    _ -> Nothing

reduce mem (CaseStmt t0 x1 t1 x2 t2) =
  case (reduce mem t0) of
    Nothing -> Nothing
    (Just (UnOp "INL" u0)) -> reduce mem (subst x1 u0 t1)
    (Just (UnOp "INR" u0)) -> reduce mem (subst x2 u0 t2)
    _ -> Nothing

reduce mem (Appl t1 t2) =
  case (reduce mem t1) of
    Nothing -> Nothing
    (Just (Lam x0 t0)) -> reduce mem (subst x0 t2 t0)
    _ -> Nothing

reduce mem (LetStmt x1 t1 t2) =
  case (compute mem t1) of
    Nothing -> Nothing
    (Just c1) -> reduce mem (subst x1 c1 t2)

reduce mem t = Just t -- head normal forms: numbers, pairs, abstractions.
```

```
-- compute mem t = try to reduce t to a normal form
-- under the definitions in mem

compute :: [(String, Tree)] -> Tree -> Maybe Tree

compute mem t =
  case (reduce mem t) of
    Nothing -> Nothing
    (Just hnf) ->
      case hnf of
        (Num n) -> Just (Num n)
        (Pair t1 t2) ->
          case (compute mem t1) of
            Nothing -> Nothing
            (Just c1) ->
              case (compute mem t2) of
                Nothing -> Nothing
                (Just c2) -> Just (Pair c1 c2)
        (UnOp op Diverge) ->
          if op == "ABS"
            then Nothing
            else Just (UnOp op Diverge)
        (UnOp op t) ->
          case (compute mem t) of
            Nothing -> Nothing
            (Just c) -> Just (UnOp op c)
        _ -> Nothing

fdefs = [ ("From",
          (Lam "n" (UnOp "ABS"
                   (UnOp "INR" ( (Pair (Var "n")
                                       (Appl (FVar "From")
                                             (BinOp '+' (Var "n")
                                                        (Num 1)))) ) ))),

          ("Take",
          (Lam "n"
          (Lam "xs" (Cond (Var "n")
                          (UnOp "ABS" (UnOp "INL" Diverge))
                          (CaseStmt (UnOp "REP" (Var "xs"))
                            "l"  (UnOp "ABS" (UnOp "INL" Diverge))
                            "r"  (UnOp "ABS" (UnOp "INR"
                                   (Pair (UnOp "FST" (Var "r"))
                                         (Appl (Appl (FVar "Take")
                                                     (BinOp '-' (Var "n")
                                                                (Num 1)))
                                               (UnOp "SND" (Var "r"))))))))))),

          ("TakeTwoFromSeven",
          (Appl (Appl (FVar "Take") (Num 2))
                (Appl (FVar "From") (Num 7))))

        ]

taketwofromseven = show ( compute fdefs (FVar "TakeTwoFromSeven") )
-- output:
-- Just ABS(INR((7,ABS(INR((8, ABS(INL(@))))))))
-- (799 reductions, 1849 cells)
```

Bibliography

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, second edition, 1998.

[GH99] J. Gibbons and G. Hutton. *Proof methods for structured corecursive programs,* 1999. `http://citeseer.nj.nec.com/article/gibbons99proof.html`

[Hen87] Martin C. Henson. *Elements of Functional Languages.* Blackwell Scientific Publications, 1987.

[Jon00] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell, 2000. To be published in the NATO ASI series during 2001.

[Jun96] Achim Jung (Ed.) *Domains and denotational semantics: History, accomplishments and open problems,* 1996. `http://citeseer.nj.nec.com/article/jung96domain.html`

[Win92] Glynn Winskel. *The Formal Semantics of Programming Languages: an introduction.* The MIT Press, Cambridge, Massachusetts, 1992.