INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

A Framework for Deductive Traders of Context Information

Markus Latte

Aufgabensteller: Prof. Dr. Claudia Linnhoff-Popien
Betreuer: Michael Schiffers

Abgabetermin: 1. Dezember 2004

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

A Framework for Deductive Traders of Context Information

Markus Latte

Aufgabensteller: Prof. Dr. Claudia Linnhoff-Popien
Betreuer: Michael Schiffers

Abgabetermin: 1. Dezember 2004

Abstract

Context aware services often need derived and higher level information about users and their environments. Sensors and databases mostly offer low-level information only. The need to bridge this gap demands for refinement and ennoblement of contextual information. Therefore the concept of conventional traders known from distributed systems is adapted. Their core, typically a database, is equipped with its deductive closure.

A complete deductive closure is ineligible for practical purposes because of a fairly high runtime complexity. Therefore a representation is used which is simple enough to maintain the deductive closure but which is still strong enough to model our environment adequately.

All three key functionalities of conventional traders offered at their interface are emulated, namely to add and to withdraw service offers and context information as well as to enquire the availability of a, maybe derived, service.

Algorithms for adding and removal are efficient as far as time and memory accesses are concerned since they are polynominal with low degree and measured in those changes that are imposed by deductive closeness. This avoids complete and expensive rebuildings of closures.

Inquiries for services can be treated efficiently as well. For inquiries the requester needs to provide a construction strategy and a representation of the answer. This differs from conventional traders since structurally multiple answers are possible due to deductive closeness.

Zusammenfassung

Kontextabhängige Dienste benötigen abgeleitete Informationen und Informationen auf höherem Abstraktionsniveau über ihre Benutzer und ihre Umgebung. Sensoren und Datenbanken können aber meist nur rohe und unbearbeitete Daten liefern. Die Notwendigkeit, diesen Unterschied zu überbrücken, verlangt nach Verfeinerung und Veredelung von Kontextinformationen. Dazu wird das Konzept der Trader aus dem Umfeld Verteilter Systeme herangezogen. Deren Zentralkomponente, im wesentlichen eine Datenbank, wird um ihre deduktive Hülle erweitert.

Wegen der hohen Laufzeitkomplexität eines vollständigen deduktiven Abschlusses kommt für die Praxis nur ein Fragment in Frage. Aus diesem Grund wird eine Darstellung verwendet, die einfach genug ist, um den deduktiven Abschluss aufrecht erhalten zu können, aber mächtig genug ist, um unsere Umgebung angemessen zu modellieren.

Die drei Funktionalitäten für konventionelle Trader zum Hinzufügen von Dienstangeboten bzw. Kontextinformationen, deren Entfernung sowie Anfragen nach der Verfügbarkeit von, möglicherweise abgeleiteten Diensten nachgebildet.

Die Algorithmen zum Hinzufügen und Entfernen von Diensten sind effizient, in dem Sinne, dass Laufzeit und Speicherzugriff polynominell sind in der Größe der tatsächlich notwendigen Veränderungen, welche die deduktive Abgeschlossenheit auferlegt. Dadurch werden komplette und damit auch kostspielige Neuberechnungen vermieden.

Ebenso effizient können Anfragen nach Diensten behandelt werden. Dazu muss der Anfrager im Gegensatz zu konventionellen Tradern zusätzlich eine Konstruktionsstrategie sowie eine Repräsentation der Antwort im Vorfeld angeben. In diesem Punkt unterscheiden sie sich von konventionellen Trader, da strukturell verschiedenartige Antworten aufgrund der deduktiven Abgeschlossenheit möglich sind.

Acknowledgements

This piece of work does of course not arose in complete isolation from the rest of the world. Therefore I would like to thank those who had strong influences. As it is unsuitable to balance I prefer to arrange the names alphabetically.

- Klaus Aehlig for the many discussions we had, for proofreadings, and his encouragements.
- Professor Claudia Linnhoff-Popien for discussions and her initial topic proposal. During the last two years the content changes smoothly but still with her continuous support and interest.
- Michael Schiffers for discussions, proofreadings, and comments on the literature. He accepted and supported the development to a more abstract representation as well.
- Professor Helmut Schwichtenberg for his lectures on logics. The tracks of deduction and the lambda calculus are highly visible.
- Finally, I would like to thank my other friends and my family for their benevolence and their support to finish this work. And, yes, I indeed do know the banished word[1] of the last two years: “FoPra“ …

[1] in German: Unwort

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Nowadays computerised facilities become more and more ubiquitous. The vision of pervasive computing as first expressed by Weiser [Weis 91] is turning into reality. This development has serveral causes. One of them is miniaturisation of electronical equipment. Another one is their decreasing cost of production [Moor 65].

Computer science is influenced by these developments as it has to fulfill increasingly demanding tasks requiring more processor power, memory capacity, reduced consumption of energy, and network connectivity, to name just a few.

But the main responsibility of computer scientists, however, is to design systems supporting human users in achieving their goals. The link between these aspects of computer science is a sophisticated form of interaction in order to achieve efficient communication. Programs often need information about their users in order to perform their designated tasks. The quality of information needed has to be the higher the more these systems are required to be autonomous, intuitive and convenient. This necessity is often referred to as *context awareness* [DAS 01]. Exemplary scenarios (cf. [ChKo 00]) combine qualitatively different kind of information.

- Raw data of low level. These might be measured by sensors. Examples of such data include temperature, air humidity, time, position, and velocity. Other raw data might be stored in databases and could, for example, include textual descriptions of the appendant entity, preferences, and history of former contexts. In the literature these data are called mostly *context information* (cf. [ChKo 00] or [KSB 02]) - which is owned by an *entity*.

- Information about *service providers*. These offer to convert information or to enrich contexts by deduced information. Examples of information conversions include position transformation functions, and information desks. Such deduced information can reflect situations of users for instance.

Following [KSB 02] both kinds are subsumed under *content provider* since both can be treated as functions. The first as function of zeroth order and the latter with a degree greater than zero. SML (see `http://www.smlnj.org/`) goes an akin way, it considers functions as syntactical sugar. For example the function declaration

```
fun foo x = x;
```

is internally translated into a variable declaration like

```
val foo = fn x => x;
```

To eclipse computerised equipments into the background of automatisation they need to be consolidated and to synergy. This conforms to the trader concept in distributed systems (cf. [LP 03, chapter 7] and figure 1.1) where service providers are asked to register their service offers at a central component - the so called trader. Moreover, they are allowed to withdraw their offers as well. Inquiries of *users* (or synonymously *clients* – independently whether they are human or parts of software or even *importer* - as common in distributed systems) for services are directed to a trader. The trader looks up its database for matches and answer accordingly to the caller who can finally get into touch with the proposed service providers.

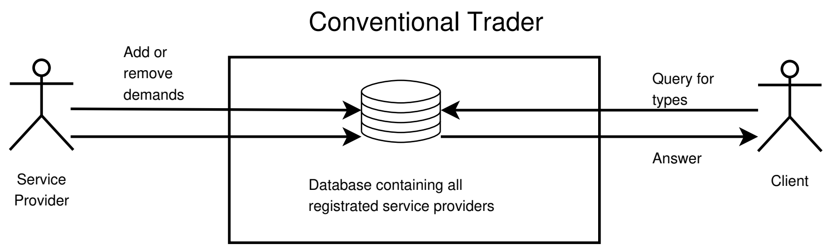


Figure 1.1: Main components of and data flow within a conventional trader.

The common lookup corresponds to a simple selection operation on a relation in a database system. This suffices for most systems looking for something usable like yellow pages, but suffers from a fairly obvious drawback: it is not deductively closed.

As an example consider the conversant but simplified problem to compose services: Given a `.tex` file, say `file`, a service `latex` able to construct `.dvi` files out of this and a further service `dvips` able to convert the latter into postscript. The user might ask to build a postscript out of its `.tex` source. Suppose all these information - but not more - are entered into the trader system and the user queries for such a composition. As expected the answer is "no" since no immediate service for the type "postscript" was announced. To elude this unwished result two adaptations are required.

- Firstly, the structure of possible answers needs to be amplified. But how should an answer look like? In our example it should express that in order to achieve a postscript file the `.tex` source should be applied to the first service and then its result to the second.

  As a generic solution we choose *simple typed lambda terms* since functional languages avail them and form an excellent playground.

  In our actual case we could form an answer expression as a function composition like:

  $$\mathtt{dvips}^{\mathtt{dvi}\to\mathtt{ps}}(\mathtt{latex}^{\mathtt{tex}\to\mathtt{dvi}}\mathtt{file}^{\mathtt{tex}}),$$

  whereby types are written as superscript.

  Even in this simple example we can use SML as representative of functional languages as a test enviroment primarily for type checking and normalisation (cf. chapter 2).

  The user can understand the answer as a description of how to construct the desired information: replace free variables by their representing service functions and normalise the resulting term which means to execute their beta redexes.

  Hereby service providers are assumed to be omnipresent - at least at the client. As often services are available only at some specialised network nodes, terms could also be translated into messages containing instruction sequences and lists of values. These can be sent around in the network like tokens or software agents. As an example consider $g^{B\to C}(f^{A\to B}a)$ and assume that the client owns a, f is located at a node called F and g at G. The user could initiate the computation by sending a message $\langle$`codeF`, [value of a]$\rangle$ to node F whereby `codeF` is

apply first argument to f and save result at aux;
send a message $\langle$`codeG`, $[aux]\rangle$ to G whereby
`codeG` is apply first argument to g and save result at aux;
send a message $\langle$ $\boxed{\text{this is the result}}$, $[aux]\rangle$ to client

The user might take advantage of parallel computation if the term is long and not too nested.

- Besides the linguistic extension the "brain" of traders needs also to be improved which is the main topic of this thesis. For the subsequent we refer to figure 1.2.

  As a database cannot act deductively by itself its deductive closure is provided to the trader, representing all possible service combinations. To get the complexity under control service combinations are not materialized explicitly and the closure is not completely rebuilt due to update operations but is rather updated on-the-fly. We will tackle the deductive closure in chapter 4 complying with the elementary operations `add` and `remove`, cf. definition 39.

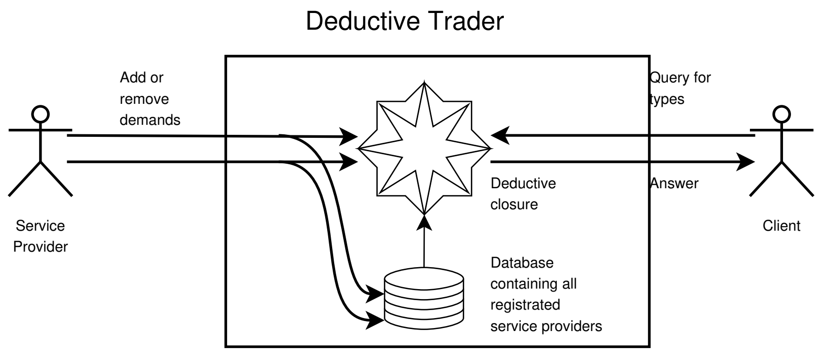


Figure 1.2: Main components of and data flow within a deductive trader.

  Therefore inquiries by users are redirected from the database to its closure. Demands for updates are directed to both. The database will only hold additional information not used within the closure like, or information about vendors. The database is also used to prevent that identical services occur with different types since the closure can be understood as being indexed only by types whereas the database can be additionally indexed by names.

As an integration into an existent and enclosing framework should be possible the interface of conventional traders is reused for the proposed deductive trader:

- adding service offers (by content providers),
- withdrawing service offers (by content providers), and
- queries for a certain service (by clients).

1.2 "Types" types the world

More than for conventional trader our approach focuses on types of information as a complete description of their characteristics. Consequently information of equal type are considered as comparable to each other and thus treated as equal. Among others, this type driven approach bases on the famous slogan (cf. [Miln 78] and [MyO' 84])

Well-typed programs cannot go wrong.

In contrast to those services handled by conventional traders context information handled by deductive traders are treated without any kind of dynamic or static parameters usable for inquiries. This restriction stems from function composition itself. For example consider *costs per page* being affixed on printer services. Assume a typesetting service allowing to convert text equipped with visual markups into a typeset and further a printing service taking the latter as input. As the amount of pages produced by the typesetting programme is not predictable in advance, an information about the costs per page is almost meaningless.

In other words, additional information cannot be used for inquiries within our approach so far. Thus *quality of service* (QoS for short) aspects cannot be realised directly. Nevertheless such additions can be offered as dynamic property by the content providers. If the granularity is coarse enough they can be encoded into the type system directly. Section 4.5 discusses how this can be accomplished but at the price of an increased complexity (since the corresponding *covering set* contracts, cf. definition 37). Independently, dynamic properties are incumbent upon the user to check their validity as they reside outside the scope of traders.

In general the existence of an common typing system is assumed. It should describe services and their abilities properly. The intelligence for problem solving remains at the service providers and their cooperation. Since typing in our framework resembles that of conventional traders we do not detail its design, management, etc.

To get the trader system viable, context information and service providers need to be melted. Security in the sense of value hiding is guaranteed as the traders do not notice any value. Instead they notice only its type. Types at their own do not provoke any security leak in general. For example it is more dangerous to trumpet one's bank balance than to say that someone has a bank account.

As in contrast to conventional traders our approach requires an integration of the inquirer's context. At the first glance, it seems to be inferior as the complexity to assemble the deductive closure is cubical and not linear. This can be revised if the principles of object oriented development – as a model of the surrounding world – is additionally taken into account. The difference between classes and objects plays the same role like types and values do in our approach. Even if the abundance of entities is huge the amount of entity kinds is rather small as they reflect only the diversity of products. Since traders do not perceive values they can not distinguish among different entites of identical kind.

Another pillar of object orientation is inheritance. It can be adapted for an hierarchy of entity kinds as types of context information can be handed over but not necessarily their values. As a showcase we use inheritance in figure 1.3 which is not asserted to be exhaustive. Without wanting to advertise, another more detailed branch of such modeling can be gazed at `http://www.wordreference.com/definition/computer.htm` or similar web directories.

Apart form inheritance, hierarchy can be utilized to parallise traders as indicated by figure 1.4. Each entity kind is tackled by a dedicated trader and its closure can be fed by the immediate lower (i.e. rootwards) trader.

Inquiries of clients are to be addressed to the trader for its entity kind. Service providers interact only with the root entity. Every change in the closure is to be propagated leafwards. As information could be independently deducible for two different entity kinds - one is the descendant of the other -, service compositions must be labeled with their origin since this constellation could be released by a withdrawal of an involved service provider.

Our type driven traders operate on a more abstract level than proprietary approaches of context aware application as summarised in [ChKo 00, chapter 4]. This abstraction supersedes special

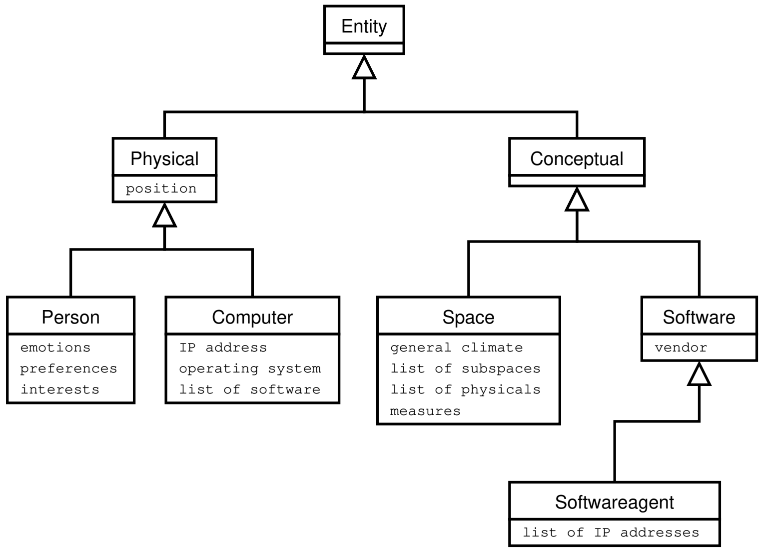


Figure 1.3: Hierarchy of entity kinds.

treatments for position, time, temperature, and so on.

To sum up, deductive trader can advance context aware services.

1.3 Requirements and Restriction

The requirements for a deductive trader are:

1. It should fit the role of traditional traders, i.e. should implement the interface of a trader but extend its semantics as described above. This interface consists of operations for adding and removal as well as a function for inquiries.

2. Both adding and removing of content providers should be arranged efficiently. In particular, the deductive closure should not be rebuilt because of these *update operations*, instead only mandatory parts should be realigned in polyominal time (with low degree) measured in their cardinality.

3. It should be possible to gather update demands to adjust adequately individual costs for updates. The decision when the necessary operations take place are not detailed in this thesis.

4. As changes of types effect the closure rather than changes of their values, updates are presumed to be less frequent than inquiries. Therefore update operations should maintain data structures to speed up inquiries, more precisely some kind of calculation in advance.

Nevertheless this thesis should be treated as a proof of concept rather than an ingenious, detailed and straightly practical system. Therefore we need to exclude some aspects as

- replication of traders,
- hierarchies of entities as above mentioned,
- dynamic service attributes, as for example current queue length, operational capacity and so forth,
- every kind of management, and

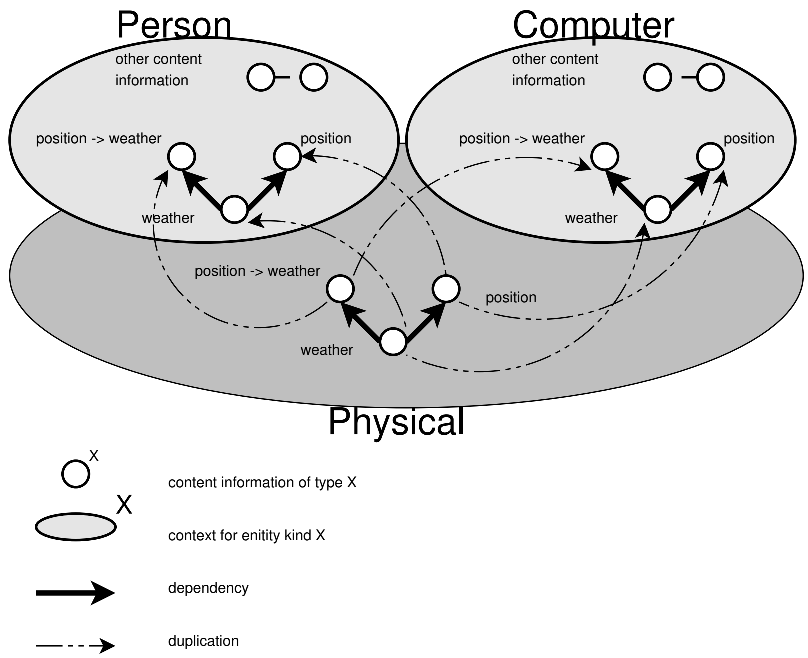


Figure 1.4: Inheritance as means for parallelising.

- every kind of organising types.

1.4 Specification

The trader administrates a set X of content providers and maintains its deductive closure X^*. The asterisk denotes the closure, a concept formally defined in definition 30.

The management operations for adding and removal of providers are realised at the level of sets just by union and difference and at the level of its deductive closure by $\texttt{add}(\Box, \Box)$ and $\texttt{remove}(\Box, \Box)$. These are executed simultaneously on both structures. In the case of the closure some efforts are to be undertaken in chapter 5 and 6 to get following diagrams commutative:

$$
\begin{tikzcd}
X \arrow[r, "\Box\cup R"] \arrow[d, "*"'] & \Box \arrow[d, "*"] \\
\Box \arrow[r, "\texttt{add}(\Box{,}R)"] & \Box
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[r, "\Box\setminus R"] \arrow[d, "*"'] & \Box \arrow[d, "*"] \\
\Box \arrow[r, "\texttt{remove}(\Box{,}R)"] & \Box
\end{tikzcd}
$$

The topmost lines characterise the behaviour of traditional traders.

For queries we anticipate components for inquiries (cf. figure 1.5) and their interconnections and demonstrate their necessity later during this thesis.

- The inquired <u>type</u>. It has been left unchanged from the conventional traders.
- A <u>selection strategy</u> specifying how the result should be built up.
- The <u>selector</u> handles ambiguities and can access both static and dynamic parameters of content providers.

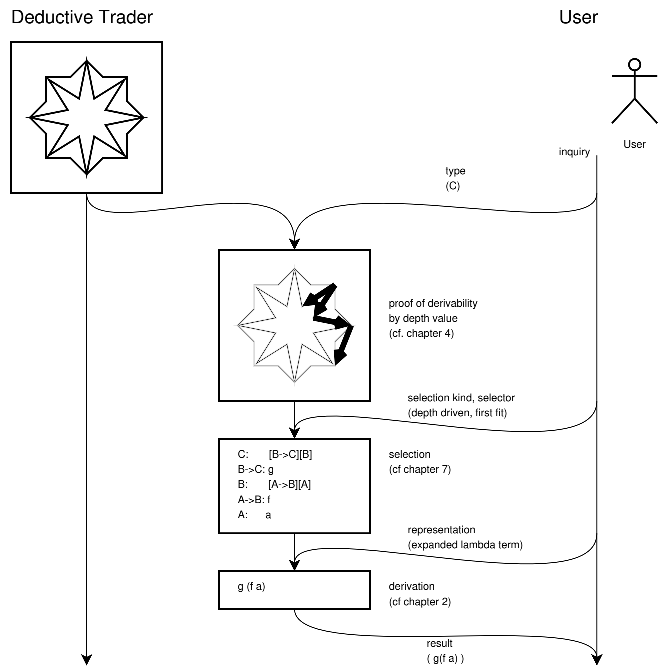


Figure 1.5: Dependency of the parts of an inquiry to each other.

- A description of the representation of the result.

All subsequent descriptions focus on the trader's point of view.

1.5 Comparison with present software

The `make`[SuS 89] **tool** is an often used tool for software development to construct files out of others and can be used for the initial question involving `latex` and `dvips`. `Makefiles` would correspond to the database of conventional traders. But it contradicts requirement four as it does not tender to compute in advance since inquires are answered by a depth first search.

On the field of **context awareness.** According to the survey article by Chen and Kotz (cf. [ChKo 00]), which was published 2000, only proprietary and specialised approaches are available so far.

Theorem provers could be used but are ineligible for reasons discussed in section 3.1 on page 17.

Logical programming languages and **deductive database systems** provide opportunities. For example Prolog [BrEi 04, chapter 5] can continue the initial example of page 2 by emulating lambda terms (cf. subsequent chapter).

```
tex(file).
tex_to_dvi(latex).
dvi_to_ps(dvips).

dvi(app(X,Y)) :- tex_to_dvi(X), tex(Y).
ps(app(X,Y)) :- dvi_to_ps(X), dvi(Y).
```

The first block represents the database and the second realises deductive closeness (cf. definition 28) for some types. The question `ps(X)` asking for a function composition to obatain postscript is properly answered by `X = app(dvips, app(latex, file))`.

To complete closeness for instance `dvi_to_ps(trivial_abs(X)) :- ps(X).` has to be added. Therefore traversing alternatives can lead to a stack overflow as cycles might occur. We meet a similar challenge in subsection 6.2.2.1 for our approach.

Even if logical languages could achieve our aim concerning presentability, they violate the second requirements on page 5 as they use mainly depth first searching.

Anyhow both influence the concept of our approach. Instead of writing down explicitly how to compute, as characteristic for imperative programming languages, they accept descriptions of facts and determinate the computation by theirselves.

Our approach takes advantage of the concept of deductive closeness and simplified theorem provers.

1.6 Outline

This chapter pointed out the necessity for deductive traders in dealing with context awareness. The subsequent chapter introduces the lambda calculus to refine the original question. Its decidability is discussed in the third chapter and advises restrictions to master its complexity. Chapter 4 concretises the initial question to a data structure called *context* and proves its optimality. From the point of view of service providers chapter 5 and 6 explain the functionalities of the elementary operations `add` and `remove`. Next, from the reverse side, the side of the enquirers, we detail how answers can be formed. Examples are presented in chapter 7.

1.7 Notations and Preliminaries

1.7.1 General Notations

- n is considered as both natural number and the set $\{0, \dots n-1\}$ like in set theory ([Deis 02, 2. Abschnitt, 6. Ordinalzahlen, "Die moderne Definition einer Ordinalzahl"; page 177]).
- $(\overline{\mathbb{N}}, \leq)$ is a *well ordering* on $\overline{\mathbb{N}} := \mathbb{N} \uplus \{\infty\}$ as an extension to $(\mathbb{N}, \leq)$ whereby $\leq$ is amplified by $\{(x, \infty) \mid x \in \overline{\mathbb{N}}\}$. Notice that $(\overline{\mathbb{N}}, \leq)$ equals $\omega + 1$. Whereby ω is the least infinite ordinal [Deis 02]. However the subsequent does not use ordinals directly.
- For two sets A and B, $A \uplus B$ equals $A \cup B$ and denotes implicitly that A and B are disjoint.
- "iff" abbreviates "if and only if".
- "wlog" abbreviates "without lost generality".
- As most of the occurring tuples are implemented as classes and their components as their members or attributes the name of components are printed in a monospace font and accesses to components are symbolised by the dot notation. For example if x is a tuple $(\mathtt{A}, \mathtt{B})$, $x = (x.\mathtt{A}, x.\mathtt{B})$ is a tautology.
- For a function $f : A \to B$, $dom(f)$ denotes its domain and $range(f)$ its range. $f : A \hookrightarrow B$ symbolises a partial map for A to B, meaning $dom(f) \subseteq A$.
- For a set X, 2^X denotes its powerset, meaning the set of all subsets of X.

- For a function $f : A \to B$, $f : 2^A \to 2^B$ denotes its extension on powerset such that

$$f(X) := \{f(x) \mid x \in X\}$$

for every subset X of A.

1.7.2 Complexity theory

We use for asymptotic behaviour of functions the common notations $\mathcal{O}$, o, Ω, ω θ, ..., cf. [Reis 90, subsection 1.3.2]. Languages are printed as normal text (e.g. SAT) whereas complexity classes in bold (e.g. **NPC**). For latter we use explicitly **P**, **NP**, **NPC**, **coNP**, **PSPACE**, and some of their oracle classes. For definitions and details confer [Reis 90, chapter 6] or [Papa 94, chapter 17 and 19].

1.7.3 Source and Pseudo Codes

The attached source code has been written in Java 1.5 as a prototype. During the chapters we use pseudo codes to concretise algorithms rather than excerpt from its implementations. We presume that no run time error takes place and variables differ from the `null` pointer.

Chapter 2

The Simple Typed λ Calculus with Pairs

A very short introduction to λ calculus is given.

2.1 Types

Definition 1. The set of *types* denoted by $\mathbb{T}$ is given by the following grammar:

$$T ::= A \mid (T \to T) \mid (T \wedge T)$$

whereby A is representative for *atomic types*.

Types are written in capital letters. For simplicity we omit outermost brackets, assume that conjunction has a higher precedence than implication, and treat chains as right associative.

Definition 2. The functions subtypes and directSubtypes : $\mathbb{T} \to 2^{\mathbb{T}}$ collect for a type its *subtypes* and its *direct subtypes* (or synonymically *immediate subtypes*) respectively. Let $\circ$ range over $\{\wedge, \to\}$.

$$\begin{aligned} \text{directSubtypes}(S_0 \circ S_1) &:= \{S_0, S_1\} \\ \text{directSubtypes}(X) &:= \emptyset && \text{(if } X \text{ is atomic)} \end{aligned}$$

$$\begin{aligned} \text{subtypes}(S_0 \circ S_1) &:= \{S_0 \circ S_1\} \ \cup \ \bigcup_{i \in 2} \text{subtypes}(S_i) \\ \text{subtypes}(X) &:= \{X\} && \text{(if } X \text{ is atomic)} \end{aligned}$$

Hereby atomic types shall represent an annotation to values about the kind of their information content like position, temperature, time, duration, user's preference, and so on. Analogically to programming languages values of the conjunction type $S \wedge T$ can be considered as an ordered pair consisting of values of type S and T, and a value of type $S \to T$ as a (total) function mapping values of type S to those of type T. These types do *not* denote types on implementational level like `integer`, `real`, or `bool` does. Indeed it is essential that the underlying typing system of the proposed framework is sufficiently granular to detail all demanded aspects of context information. Development and maintenance of a proper type system require of course an adequate management. The subsequent analysis premises such an typing system, but we disregard its details.

Definition 3. The *size* $|T|$ of a type T is:

$$\begin{aligned}|A| &:= 1\\ |T_0 \to T_1| &:= 1 + |T_0| + |T_1|\\ |T_0 \wedge T_1| &:= 1 + |T_0| + |T_1|\end{aligned}$$

The notion *length* is used synonymously for *size*.

2.2 Lambda terms and some of their properties

So far we introduced a type system. Now its inhabitants and their operational aspects are addressed.

For every type T we presume a set Var^T of infinite many variables of type T, written as $x, y, z \ldots$, possibly ornamented with super- or subscripts or primed. For two different types their sets are required to be disjoint.

To define *simple typed lambda terms with pairs* (for short: *term* or *derivation*) in the typed-term style of Curch (see [Hind 97, subsection 2A3]) we define by simultaneous induction on types auxiliary sets Λ^T of all lambda of type T:

Definition 4. The auxiliary family $\{\Lambda^T\}_{T\in\mathbb{T}}$ is inductively defined by

$$\frac{}{\mathrm{Var}^T \subseteq \Lambda^T}\ \text{(variable)}$$

$$\frac{x \in \mathrm{Var}^S \qquad t \in \Lambda^T}{(\lambda x t) \in \Lambda^{S\to T}}\ \text{(abstraction)} \qquad \frac{r \in \Lambda^{S\to T} \quad s \in \Lambda^S}{(rs) \in \Lambda^T}\ \text{(application)}$$

$$\frac{t \in \Lambda^{T_0\wedge T_1}}{(\pi_i t) \in \Lambda^{T_i}}\ \text{(for } i \in 2\text{, projection)} \qquad \frac{t_0 \in \Lambda^{T_0} \qquad t_1 \in \Lambda^{T_1}}{(\langle t_0, t_1\rangle) \in \Lambda^{T_0\wedge T_1}}\ \text{(pairing)}$$

This yields the set of typed lambda terms:

$$\Lambda := \bigcup_{T\in\mathbb{T}} \Lambda^T$$

Analogously we understand variables as

$$\mathrm{Var} := \bigcup_{T\in\mathbb{T}} \mathrm{Var}^T$$

Inversely, $\mathrm{type} : \Lambda \to \mathbb{T}$ maps a lambda term to its (unique) type. To smooth notations we say that a term t *inhabits* a type T if and only if $\mathrm{type}(t) = T$ and mean by "T is inhabited" that a term t exists with $\mathrm{type}(t) = T$.

To smooth the notation of terms we introduce some abbreviations:

- we omit outermost brackets,
- application is left associative,
- abstraction has the greatest precedence,
- a dot symbols a parenthesis pair starting at the dot position and closes rightmost,
- types annotation can be omited if determinable and
- instead of t^T we write $t : T$ if T is lengthy.

A central idea of the lambda calculus its variable binding. λxt binds all occurences of x within t. Consider as an example

$$\lambda y.zy\lambda x\lambda x.yx$$

whereby arrows emphasise bindings.

Bindings play the main part rather than their names. Consequently we treat terms as equal up to *α-equivalence*, meaning up to bounded renaming. Continuing the above example it is α equivalent to

$$\lambda y.zy\lambda x\lambda \mathbf{w}.y\mathbf{w}$$

but not to

$$\lambda y.zy\lambda \mathbf{w}\lambda x.y\mathbf{w}.$$

Equality of terms is treated up to α equivalence.

All unbound variables are covered by the set of free variables FV.

Definition 5 (free variables). The function $\mathrm{FV} : \Lambda \to 2^{\mathrm{Var}}$ on terms is defined as

$$\begin{array}{lcll}
\mathrm{FV}(x) & := & \{x\} & \\
\mathrm{FV}(\lambda xu) & := & \mathrm{FV}(u) \setminus \{x\} & \\
\mathrm{FV}(uv) & := & \mathrm{FV}(u) \cup \mathrm{FV}(v) & \\
\mathrm{FV}(\langle u, v\rangle) & := & \mathrm{FV}(u) \cup \mathrm{FV}(v) & \\
\mathrm{FV}(\pi_i u) & := & \mathrm{FV}(u) & \text{(for } i \in 2)
\end{array}$$

A term is *closed* iff its set of free variables is empty.

The main operation on terms is reduction. It presupposes *substitution*.

Definition 6 (substitution). Let x be a closed term and t an arbitrary term. The *substitution* $\Box[x := t]$ is defined by recursion on the applied term $\Box$:

$$\begin{array}{rcll}
y[x := t] & := & \begin{cases} t & \text{if } x = y \\ y & \text{otherwise} \end{cases} & \\
(\lambda yr)[x := t] & := & \lambda y.r[x := t] & (\text{wlog. } x \neq y \text{ and } y \notin \mathrm{FV}(r)) \\
(rs)[x := t] & := & r[x := t]s[x := t] & \\
(\pi_i r)[x := t] & := & \pi_i(r[x := t]) & \text{(for } i \in 2) \\
\langle r, s\rangle[x := t] & := & \langle r[x := t], s[x := t]\rangle &
\end{array}$$

Definition 7 (iterative substitution). Let W be a set of closed terms, $v : W \to \Lambda$ and t an arbitrary term. The *iterative substitution*

$$t[u := v(u)]_{u \in W}$$

is defined by recursion of W as

$$\begin{array}{ll}
t[u := v(u)]_{u \in \emptyset} & := t \\
t[u := v(u)]_{u \in W \uplus \{w\}} & := t[w := v(w)][u := v(u)]_{u \in W}
\end{array}$$

The independence of the iteration orders for W is silently presumed.

Definition 8 (reduction). Let s, t, t_0, t_1 be terms and x a variable. The (one step) reduction $\mapsto$ is defined by $\mapsto_\beta \cup \mapsto_\pi$.

$$\begin{array}{rcll}
(\lambda xt)s & \mapsto_\beta & t[x := s] & \\
\pi_i\langle t_0, t_1\rangle & \mapsto_\pi & t_i & \text{(for } i \in 2)
\end{array}$$

The terms on the left hand side are called (β-, π-)*redexes*. Its continuation to the relation $\to_\beta$[1] is:

$$\frac{t \mapsto_\beta t'}{t \to t'} \qquad \frac{t \mapsto_\pi t'}{t \to t'}$$

$$\frac{t \to t'}{st \to st'} \qquad \frac{t \to t'}{ts \to t's}$$

$$\frac{t \to t'}{(\lambda x.t)s \to (\lambda x.t)} \qquad \frac{t \to t'}{\pi_i t \to \pi_i t} \quad \text{(for } i \in 2)$$

$$\frac{t \to t'}{\langle s,t\rangle \to \langle s,t'\rangle} \qquad \frac{t \to t'}{\langle t,s\rangle \to \langle t',s\rangle}$$

whereby the corresponding types are omitted and rules apply only if they conform to the definition of lambda terms. $\to_\beta^+$ and $\to_\beta^*$ denoted the transitive respectively the transitive reflexive closures of $\to_\beta$.

Definition 9. Let s be a λ term.

- s is called *normal* if no term t with $s \to_\beta t$ exists.
- s is called *strong normalising* if no infinite sequence $(s_i)_{i\in\mathbb{N}}$ with $s_0 = s$ and $s_i \to_\beta s_{i+1}$ for all i exists.
- s is called *confluent*[2] if for all terms t_0 and t_1 with $s \to_\beta^* t_0$ and $s \to_\beta^* t_1$ a term t' exists such that $t_0 \to_\beta^* t'$ and $t_1 \to_\beta^* t'$.

Lemma 10. Every term is strong normalising and confluent.

Proof. Consider [TrSc 96, chapter 6] and adapt definition 1.2.12 and theorem 1.2.15 in [TrSc 96]. □

Definition 11. For every term t its <u>unique</u> normal form is $\downarrow t$.

Note that the normal form is unique because of strong normalisiation and confluence.

Thus equality can be defined as

Definition 12. Two terms s and t are equal (in sign $s =_\beta t$) if and only if $\downarrow s = \downarrow t$.

Definition 13 (depth). The function depth : $\Lambda \to \mathbb{N}$ is defined as

$$\begin{array}{lcll}
\text{depth}(x) & := & 1 & \\
\text{depth}(\lambda xu) & := & 1 + \text{depth}(u) & \\
\text{depth}(uv) & := & 1 + \max(\text{depth}(u), \text{depth}(v)) & \\
\text{depth}(\langle u,v\rangle) & := & 1 + \max(\text{depth}(u), \text{depth}(v)) & \\
\text{depth}(\pi_i u) & := & 1 + \text{depth}(u) & \qquad \text{(for } i \in 2)
\end{array}$$

Definition 14 (size). The function size : $\Lambda \to \mathbb{N}$ is defined as

$$\begin{array}{lcll}
\text{size}(x) & := & 1 & \\
\text{size}(\lambda xu) & := & 1 + \text{size}(u) & \\
\text{size}(uv) & := & 1 + \text{size}(u) + \text{size}(v) & \\
\text{size}(\langle u,v\rangle) & := & 1 + \text{size}(u) + \text{size}(v) & \\
\text{size}(\pi_i u) & := & 1 + \text{size}(u) & \qquad \text{(for } i \in 2)
\end{array}$$

[1] The proper notation would be $\to$. But as we use $\mapsto_\pi$ only in some proofs and focus more on $\mapsto_\beta$ the subscript β is suffixed.

[2] More precisely not a term itself is confluent but $\to_\beta$ is.

Definition 15 (subterm). The function subterms : $\Lambda \to 2^{\Lambda}$ is defined as

$$
\begin{array}{lcll}
\text{subterms}(x) & := & \{x\} & \\
\text{subterms}(\lambda xu) & := & \{\lambda xu\} \cup \text{subterms}(u) & \\
\text{subterms}(uv) & := & \{uv\} \cup \text{subterms}(u) \cup \text{subterms}(v) & \\
\text{subterms}(\langle u, v\rangle) & := & \{\langle u, v\rangle\} \cup \text{subterms}(u) \cup \text{subterms}(v) & \\
\text{subterms}(\pi_i u) & := & \{\pi_i u\} \cup \text{subterms}(u) & (\text{for } i \in 2)
\end{array}
$$

Definition 16 (Abbreviations). Let Γ be every finite set of variable and T a type.
$\Gamma \vdash t : T$ and $\Gamma \vdash t^t$ abbreviate $\text{FV}(t) \subseteq \Gamma$.
$\Gamma \vdash^? T$ (*derivability problem* for short) holds if and only if a term t with $\Gamma \vdash t : T$ exists.

2.3 Comparison to context awareness

So far we can compare the λ calculus with concepts of context awareness:

λ calculus	context awareness
free variable	context information
term	nondeterministic computation instruction
reduction order	computation trace
type	classification of values
implication type	function
conjunction type	data pairing

The original requirements for deductive traders can now be reformulated. Service offers can be understood as variables. Let C denote the set of all offers registered at the trader. An inquiry for a type T is translated into the question whether a term t exists with $C \vdash t : T$. In the positive case such a term should be returned.

2.4 Implementation

Types are realised within the package `type`. Every kind of type constructor corresponds to some class either `TypeAtom`, `TypeImpl` or `TypePair`. Each is derived from the class `Type` harbouring some elementary queries for instance for subtypes.

Lambda term constructors are analogously derived from `Lambda` within the package `lambda`. Among other things it performs substitution, but the caller bears the responsibility to avoid cyclic term representations. As the behaviour of bounded and free variables differ, they are implemented in differend classes `LambdaBound` and `LambdaVar`. Depth values are implemented as `int`. As the definition 27 of $\text{depth}_{\mathbb{C}}$ later on allows the value ∞, `Integer.MAX_VALUE` is treated as ∞.

Chapter 3

Decidability, Complexity and Restrictions

So far we know how to translate inquiries into the problem of deducibility ($C \vdash t : T$). But is this question decidable at all, meaning can an algorithm exist which solves this problem? If so, what is its time complexity? Is it efficiently solvable?

3.1 Decidability

Strong normalisation turns the black box of a proof into a white one. More formally it leads to the

Lemma 17 (subformula property). Let t be a normal term with $C \vdash t : T$. The type of every subterm of t is a subtype of T or of the type of some variable in C.

Proof. Adapt the proof of theorem 6.2.6 in [TrSc 96] and consult the CURRY-HOWARD isomorphism in [Hind 97]. □

As the amount of subformulae is finite deducibility is decidable.

Lemma 18. An algorithm exists deciding constructively whether a term t exists with $C \vdash t : T$.

Proof. Decidability corresponds to theorem 4.2.5 in [TrSc 96] for the GENTZEN calculus. Chapter 3.3 in [TrSc 96] and the CURRY-HOWARD isomorphism mediate between GENTZEN and the simpled typed lambda calculus. □

Fortunately our aimed goal is decidable. On the first glance one might suggest a *theorem prover* (*prover* for short) specialised in minimal or even in intuitionistic[1] logic utilizing the CURRY-HOWARD isomorphism once more.

To return a proper term to the inquiring user the knowledge of its existence per se is not sufficient. Hence a prover should also offer to return a proof witness, which should be convertible into a lambda term within an linear amount of time.

For example `PVS` (see `http://pvs.csl.sri.com/`) drops out as it does not provide proof objects equaling lambda terms. Indeed it uses proof objects, but organised as a tree consisting of applied tactics. Thus an internal proof of an automated proof using e.g. the `prop` or `bddsimp` instruction is only atomic and hence useless for us. Additionally as these proofs are classical it can "prove"

[1] As we do not use $\perp$ as an atomic type intuitionistic logic is conservative over minimal logic.

statements unprovable in minimal logic, consider for example the PIERCE formula [Hind 97, subsection 6A1]: $((A \to B) \to A) \to A$.

Even if a prover could return lambda terms most provers present them in normal form since the subformula property affords a decision methode. As demonstrated in section 7.2 the gap between normal terms and minimal terms might be exponential and even exponential in the size of C for the decision problem $C \vdash^? T$. To the knowledge of the author no current prover produces minimal terms.

Albeit one might argue that these cases are rather rare, but conventional provers suffer from additional disadvantage. They are not designed to benefit from multiple invocations as they tend to be applied to empty contexts. But this clashs the fourth requirement on page 5.

3.2 Complexity

We show that the derivability problem $\vdash^?$ is complete for **PSPACE** (for details on this class consider [Reis 90, section 6.4.2] or [Papa 94, chapter 19], used notation is borrowed from the latter).

Definition 19 (QBF, QSAT). QBF (*Quantified Boolean Formulae*) is the set of all second-order propositional closed formulae in prenex and rectified normal form. Explicit conjunctions, disjunctions, negations, and implications are allowed. QSAT is the subset of QBF containing precisely those formulae which are valid in common sense.

Lemma 20. Derivablitity is **PSPACE** complete.

Proof. Following [dS 76] derivablitity is in **PSPACE**.
As QSAT is complete for **PSPACE** (cf. [Papa 94, theorem 19.1]) it suffices to show $QSAT \leq_p \vdash^?$. Let F be a quantifed boolean formula. We translate F within polynominal time into a question of derivablitity: $\Gamma_F \vdash^? S_{F,1}$. Hereby Γ_F consists exactly of:

- for every $\exists x \phi \in Sub(F)$:

$$c_{\exists x\phi,1,w} : (S_{x,w} \to S_{\phi,1}) \to S_{\exists x\phi,1} \qquad \text{(for } w \in 2\text{)}$$

- for every $\forall x \phi \in Sub(F)$:

$$c_{\forall x\phi,1} : (S_{x,0} \to S_{\phi,1}) \to (S_{x,1} \to S_{\phi,1}) \to S_{\forall x\phi,1}$$

- for every $\neg \phi \in Sub(F)$:

$$\begin{aligned} c_{\neg\phi,1} &: S_{\phi,0} \to S_{\neg\phi,1} \\ c_{\neg\phi,0} &: S_{\phi,1} \to S_{\neg\phi,0} \end{aligned}$$

- for every $\phi \wedge \psi \in Sub(F)$:

$$\begin{aligned} \text{left}_{\phi\wedge\psi,0} &: S_{\phi,0} \to S_{\phi\wedge\psi,0} \\ \text{right}_{\phi\wedge\psi,0} &: S_{\psi,0} \to S_{\phi\wedge\psi,0} \\ c_{\phi\wedge\psi,1} &: S_{\phi,1} \to S_{\psi,1} \to S_{\phi\wedge\psi,1} \end{aligned}$$

- for every $\phi \vee \psi \in Sub(F)$:

$$\begin{aligned}
\text{left}_{\phi\vee\psi,1} &: S_{\phi,1} \to S_{\phi\vee\psi,1} \\
\text{right}_{\phi\vee\psi,1} &: S_{\psi,1} \to S_{\phi\vee\psi,1} \\
c_{\phi\vee\psi,0} &: S_{\phi,0} \to S_{\psi,0} \to S_{\phi\vee\psi,0}
\end{aligned}$$

Whereby $Sub(\phi)$ denotes the set of all (immediate or not) subformulae of ϕ analogously to subformulae of types. Above described transducer requires obviously polynomial time.

The principal idea is to enroll subformulae of F together with a flag whether the subformula should be valid or not. Variable assignments are expressed by nested implications. Within normal terms every single assignment of a variable x to its value b corresponds to a lambda abstraction for a type $S_{x,b}$. This abstraction value can be used to form the value of subformulae. For existentially quantified formulae the abstractions are nested whereas for univerally the abstractions are abreast.

As QBF are required to be in prenex normal form and we seek for tautologies only quantifiers with a flag for valid are to be encoded in Γ_F.

Since F must be rectified, here: every quantifier has it own variable, occultation needs not to be regarded. Otherwise above translation would not respect the borders of the both problems QSAT and $\cdot \vdash^? \cdot$: $F := \exists x \exists x. x \wedge \neg x$ is false but $\Gamma_F \vdash S_{F,1}$ would hold testified by

$$c_{F,1,1}\ \lambda y^{S_{x,1}}.c_{\exists x.x\wedge\neg x,1,0}\ \lambda z^{S_{x,0}}.c_{x\wedge\neg x,1}y.\ c_{\neg x,0}z$$

$\square$

We saw that the inherent complexity is established on compelled decisions emitting their witnesses into the context. Hence we pursue the pragmatic approach to interdict context modification along the derivational tree.

Someone could argue against that this high complexity by pointing at the exponential growing proof length in case of an extensive usage of universally quantified variables. The satisfiability problem (see [Reis 90, section 6.3.1]) can refute this argumentation since in the positive case a linear derivation exists whereas deciding satisfiability might consume superpolynomial amount of time (again on the global assumption that $\mathbf{P} \neq \mathbf{NP}$).

Every local choice between $c_{\exists x\phi,1,\mathbf{0}}$ and $c_{\exists x\phi,1,\mathbf{1}}$ for some formula ϕ become a global matter since a decision can emit its witness into the proof context which might be used leafwards and can therefore affect the global structure inherently.

3.3 Exigence for fragments

The first enthusiasmus about the decidability is curtailed by its complexity. As we can not presume that the polynomial hierarchy (cf. [Papa 94, chapter 17]) collapses, there is no efficient (in the sence of $\mathbf{P}$) algorithm for it.

This brings up a barely studied problem: vulnerability by complexity.

The trader system would face the problem of break down due to the extremely time consuming computations it offers. A denial of service attack could easily be achieved by using lemma 20 and the QSAT benchmark library[2].

Even restriction of the query length can be circumvented by an adequate conspiracy among service providers and users.

[2] available at `http://www.qbflib.org/benchmarks.html`

Almost every algorithm in distributed systems bases on data transfer between two network nodes (Base64- and CRC32-coding, cf. [Jung 95, section 2.2]), common internet protocols HTTP, FTP, SMTP, POP3, IMAP, NNTP cf. `http://www.rfc-editor.org/rfcxx00.html`). Thus its complexity is only linear. Algorithm with higher time complexity are either broken down (for example blockwise usage as by SSH) or only available for the whole system (for example routing) and does not provide any point of attack for abuse as their methods are not universal enough. For example RSA (cf. [Stin 95, chapter 4]) offers primarily efficient arithmetical functions. Recall that we do not deal with attacks undermining authentication or message integrity protocols.

Only one hint to parasitic computing could be found by the author: [BFJB 01], [Bara 01], or `http://www.nd.edu/~parasite/`. Parasitic computing subsumes attacks by abuse features in a not intended manner besides exploitations of implementational mistakes or denial of service attacks. The authors of the above essays describe how the checksum algorithm of the Transmission Control Protocol could be abused for verifying assignment of boolean formula and demonstrate how the SAT question could be solved by spreading the assignment space out to different network nodes and enforce them to verify an assignment.

Although they ignore that the construction of proper IP packets consume more time than to verify the corresponding assignment and thus they break a fly on the wheel, they do point out indeed a new way of abusing service offers (here: web servers), maybe for the first time. As the capabilities of current offers is fairly limited such kind of attacks are not profitable and hence are pretty rare. Nowadays invaders focus rather on security leaks to hijack a whole computer and their abilities.

To sum up, vendor of such a trader system would offer most vulnerable system even if its implementation is sound.

An alternative is to restrict crudely the amount of time given to an algorithm to answer the inquiry. But how should a user react if she receives a time out message? She can do nothing as nothing useful can be deduced. Consider a proof searcher working with exhaustive search by testing each rule to apply. Assume that the trader knows

$$\begin{aligned} a_0 &: A_0 \\ f_i &: A_{i-1} \to A_i && \text{(for } 0 < i \leq n\text{)} \\ a_{n-1} &: A_{n-1} \end{aligned}$$

(whereby n is assumed to be sufficient large) and a user asking for A_n. Consider following trace (informal):

search a term of type A_n,	choosing f_n,	to continue with A_{n-1},
search a term of type A_{n-1},	choosing f_{n-1},	to continue with A_{n-2},
search a term of type A_{n-2},	choosing f_{n-2},	to continue with A_{n-3},
$\vdots$		
search a term of type A_i,	time out.	

In this case the trader was on the wrong track. If he had chossen a_{n-1} instead of f_{n-1} the user would have received the positive answer $f_n a_{n-1}$ in time. All in all the user can not infer anything reasonable from a time out message.

3.4 Restriction on derivations

As we assume $\mathbf{P} \neq \mathbf{PSPACE}$, no efficient (in the sense of polynominal time) programme can decide whether a term of given type whose free variables are covered by a given set exists. To achieve our goal anyhow we have to lower our sights. The concept of "derivations" needs to be rethought in such a way that all derivations can be constructed by algorithm running in lower polynominal time.

The intended usage of the proposed system and its type modeling provide incentives.

1. **Left recursive types are unusual.** For example a service of type $(A \to B) \to C$ uses another service of type $A \to B$ as an oracle. This means that the former is allowed to use the latter an unlimited amount of time. It is essential to distinguish between this setting and a situation within the type of the first service is replaced by $B \to C$. Hereby only (at most) one information of type B is claimed and plays the part of a service composition.

2. **Services are means to an end.** In real life users seeking for services do not aim to get the services for theirselves rather for an application of their own data to this service. Take for example a time table inquiry for rail connections. Generally one wishes to get a certain question (maybe having alternative dates up its sleeve) answered and not the whole time table.

Both offer a restriction on normal forms to those which do not contain proper lambda abstractions. More precisely the first imposes that no proper innermost abstraction is required as no proper service composition can be used as an oracle for another service. For example $h(\lambda x.f(gx))$ is unwished. The second connotes that outermost proper abstractions are needless. Thus following definition is motivated:

Definition 21. A term t equaling λxr is a *proper (lambda) abstraction* if and only if $x \in \mathrm{FV}(r)$ and otherwise it is called *trival (lambda) abstraction*. The latter is emphasised by the notation $\lambda^0 xr$.
A term t is *context stable* if and only if its normal form does not contain any proper lambda abstraction.
Let C be a finite set of variables. $C \vdash_{cs} t : T$ iff $C \vdash t : T$ and t is context stable.
$C \vdash^?_{cs} T$ iff a term t with $C \vdash_{cs} t : T$ exists.

The definition relies on normal forms as the subformula properties is used in next chapter for deductively closed contexts.

Herewith the initial asked question $C \vdash^? T$ can be refined to $C \vdash^?_{cs} T$.

Chapter 4

Context

4.1 Motivation

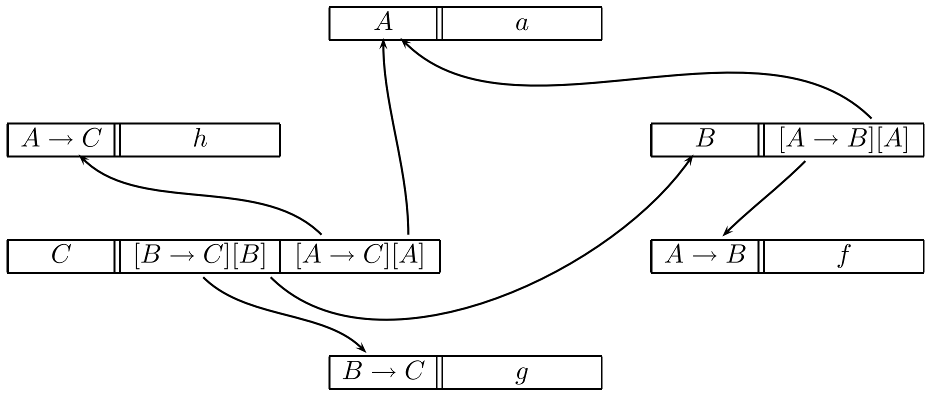
A deductively closed context for a finite set C of variables should not contain all possible derivations t with $C \vdash_{cs} t$, since the amount of such terms t is exponential in the cardinality of C (cf. the subsequent lemma 73). Thus, we need to compromise about the multivaluedness of types.

Only the type fixes the intended information. This centralisation permits to substitute an arbitrary subterm of a derivation by another covering the same set of free variables and of the same type. Thus we can *not* handle quality of service at this state. Nevertheless one can furnish this system by quality of service measures. These could be attached to variables offered by their corresponding content providers and be homomorphically extended on other term constructor rules. But be warned: this can lead to more necessary updates due to managemant operations than without, since the corresponding covering set reduces, cf. definition 37 and lemma 38.

Thus we focus on types rather than on derivation since terms are treated as equivalent if they coincide in their types. This suggests to fractionalise potential derivations into handy fragments merged by their types. These aggregations are called *contextitems*. As these snippets are not covered by the definition of lambda terms, their endings are caped by elementary stub terms (see definition 23). These terms are built by exactly one application of the construction rules in definition 4, but applied to stubs rather than to its inductive predecessor.

This suggests a system consisting mainly of items each representing a type and covering these elementary stub terms.

To detail more informally we consider exemplarily a simplified deductively closed context containing services a^A, $f^{A\to B}$, $g^{B\to C}$, and $h^{A\to C}$.



The boxes symbolise the items for the type on its left hand side. The list on the right side corresponds to its set of dervations. Brackets symbolise stubs and their links target at the corresponding contextitems.

In contrast to the later used definitions of context (cf. defintion 25) and deductive closeness (cf. defintion 28) we simplify both for the moment.

If we are for example interested in a derivation of type C we just pick up one mentioned derivation of this type and proceed by successively tracking its links until a derivation was selected without any further links. The demanded derivation is obtained by pluging a these collect derivations together.

type	choosen derivation	hence to follow
C	$[B \to C][B]$	$B \to C$ and B
$B \to C$	g	none
B	$[A \to B][A]$	$A \to B$ and A
$A \to B$	f	none
A	a	none

and leads to following derivation of type C:

$$\begin{aligned}
[B \to C][B] &\rightsquigarrow g[B] && \text{(plug in for } B \to C\text{)} \\
&\rightsquigarrow g([A \to B][A]) && \text{(plug in for } B\text{)} \\
&\rightsquigarrow g(f[A]) && \text{(plug in for } A \to B\text{)} \\
&\rightsquigarrow g(fa) && \text{(plug in for } A\text{)}
\end{aligned}$$

This example prods at some open problems:

Loop awarenes. The above algorithm does not treat possible loops. If for example $f'^{B \to A}$ is added to the context and $[B \to A][B]$ as additional derivation to A one runs into a loop if $[B \to A][B]$ is choosen as derivation for A instead of the above a.
To exclude loops depth values will be assigned to context items (cf. definition 25).

Ambiguity. Instead of $g(fa)$ we could achieve ha as well.
As the system does not enumerate all possible derivations (for its reason cf. lemma 73) the user has to provide a selection strategy.

Frugalness and simplicity. Which types should occur within the context and which derivations should be replied to the enquirer? The first is answered within the definition of contexts (cf. definition 25) and the latter by definition 28. Their sufficiency will be demonstrated in lemmata 32 and 33.

4.2 Extentions of λ terms

The set of derivation stubs of a type is modeled as a data structure called *contextitem* additionally equipped with administrative information.

Firstly we extend lambda terms to tackle derivations fragments.

Definition 22. The inductive definition 4 of Λ is extended to Λ_{stub} by replacing Λ^T_{stub} for Λ^T and by adding the rule

$$\frac{}{[T] \in \Lambda^T_{\text{stub}}} \quad (\text{stub})$$

and appending

$$\begin{aligned} FV([T]) &:= \emptyset \\ [T][x := t] &:= [T] \\ \text{depth}([T]) &:= 1 \\ \text{size}([T]) &:= 1 \\ \text{subterms}([T]) &:= \{[T]\} \end{aligned}$$

The enumeration of stubs is straightforward.

$$\begin{array}{lcll} \text{stubs}(x) & := & \emptyset & \\ \text{stubs}(\lambda x u) & := & \text{stubs}(u) & \\ \text{stubs}(uv) & := & \text{stubs}(u) \cup \text{stubs}(v) & \\ \text{stubs}(\langle u, v\rangle) & := & \text{stubs}(u) \cup \text{stubs}(v) & \\ \text{stubs}(\pi_i u) & := & \text{stubs}(u) & (\text{for } i \in 2) \\ \text{stubs}([T]) & := & \{T\} & \end{array}$$

Definition 23. A term t is an *elementary stub term* if and only if t matches to one of the patterns for some types A_0 and A_1:

$$\begin{array}{ll} x & (\text{as variable}) \\ [A_0 \to A_1][A_0] & \\ \lambda^0 x^{A_0}[A_1] & \\ \pi_i[A_0 \wedge A_1] & (\text{for both } i \in 2) \\ \langle [A_0], [A_1]\rangle & \end{array}$$

A stub $[T]$ can be regarded as a pointer to a set of (stubbed) derivations of type T. To obtain a stubless term one has successively to agglutinate stubs with a corresponding term as detailed later on in chapter 7.

4.3 Contexts

Definition 24. A *contextitem* is a tupel $(\mathtt{type}, \mathtt{depth}, \mathtt{derivations})$ consisting of

- a type `type`,
- a value `depth` being either a natural number or ∞, and
- a finite set `derivations` of elementary stub terms which inhabit the type `type`.

The set of all contextitem is denoted as *ContextItems*.

Definition 25. A *context* is a (partial) map $\mathbb{C} : \mathbb{T} \hookrightarrow \mathit{ContextItems}$ such that for every type T holds

- $\mathbb{C}(T)\texttt{.type} = T$ (coincidation), and
- stubs(t) is a subset of the domain of $\mathbb{C}$ for every term $t \in \mathbb{C}(T)\texttt{.derivations}$

and its domain is finite and minimal, i.e.

$$dom(\mathbb{C}) = \{S \in \text{subtypes}(T) \mid \text{for some variable } a \in \mathbb{C}(T)\texttt{.derivations}\}.$$

For the sequencing parts coinciding is always resumed. The second item ensures that no stubs can point to types not covered by this context.

Definition 26. Let $\mathbb{C}$ be a context. Its set of *free variables* is

$$\text{FV}(\mathbb{C}) := \bigcup_{\substack{t \in \mathbb{C}(T)\texttt{.derivations} \\ T \in dom(\mathbb{C})}} \text{FV}(t)$$

And its set of *free types* is

$$\text{FT}(\mathbb{C}) := \{T \mid t : T \in FV(\mathbb{C})\}$$

Thereby the minimality property for a context $\mathbb{C}$ can be rewritten to

$$dom(\mathbb{C}) = \text{subtypes}(\text{FT}(\mathbb{C})).$$

Remark. Minimality expresses that only those types are in the domain which are really used to form relevant derivations. Of course, this does not imply that all derivation stubs used for answers are covered. Consider an inquiry for $A \wedge B$ within a context $\mathbb{C}$ whose set of free variables equals $\{a : A, b : B\}$. Here $A \wedge B$ is not in the domain of the context and thus no proper derivation is manifested in the context.

We solve this apparent problem by an not materialized extention of the domain in definition 31 and lemma 33. For our example it means to construct $\langle a, b\rangle$ according to lemma 32.

Definition 27 (depth for stubs). $\text{depth}_{\mathbb{C}} : \mathbb{T} \to \overline{\mathbb{N}}$

$$\begin{array}{lcll}
\text{depth}_{\mathbb{C}}(c) & := & 1 & \\
\text{depth}_{\mathbb{C}}(uv) & := & 1 + \max(\text{depth}_{\mathbb{C}}(u), \text{depth}_{\mathbb{C}}(v)) & \\
\text{depth}_{\mathbb{C}}(\lambda x u) & := & 1 + \text{depth}_{\mathbb{C}}(u) & \\
\text{depth}_{\mathbb{C}}(\langle u, v\rangle) & := & 1 + \max(\text{depth}_{\mathbb{C}}(u), \text{depth}_{\mathbb{C}}(v)) & \\
\text{depth}_{\mathbb{C}}(\pi_i u) & := & 1 + \text{depth}_{\mathbb{C}}(u) & (\text{for } i \in 2) \\
\text{depth}_{\mathbb{C}}([T]) & := & \begin{cases} \mathbb{C}(T)\texttt{.depth} & \text{if } T \in dom(\mathbb{C}) \\ \infty & \text{otherwise} \end{cases} &
\end{array}$$

In order to be able to access more quickly to often needed information we extend contextitems by some syntactical abbreviations (*secondary attributes* in contrast to the *primary attributes* of definition 24) for every type T within the domain of $\mathbb{C}$:

- `supertypes` as $\{S \in dom(\mathbb{C}) \mid T \in \text{directSubtypes}(S)\}$,
- `desiredDepth` as
$$\min_{t \in \mathbb{C}(T)\texttt{.derivations}} \text{depth}_{\mathbb{C}}(t)$$
- `activity` as the cardinality of
$$\{s : S \in \text{FV}(\mathbb{C}) \mid T \in \text{subtypes}(S)\}$$
- `changeObservers` as
$$\{t \in \mathbb{C}(T')\texttt{.derivations} \mid T \in \text{stubs}(t)\}$$

The central idea behind the whole trader system is its deductive closeness:

Definition 28 (Closeness). Let $\mathbb{C}$ be a context and $\mathcal{T}$ a subset of *dom*$(\mathbb{C})$. $\mathbb{C}$ is called *structurally closed with respect to* $\mathcal{T}$ iff for every $T \in \mathcal{T}$ holds:

(1) $\to^-$**:** if $T = A \to B$ and both $\mathbb{C}(A \to B)$`.derivations` and $\mathbb{C}(A)$`.derivations` are not empty then $[A \to B][A] \in \mathbb{C}(B)$`.derivations`.

(2) $\to^+$**:** if $T = A \to B$ and $\mathbb{C}(B)$`.derivations` is not empty then $\lambda^0 x^A.[B] \in \mathbb{C}(A \to B)$`.derivations`.

(3) $\wedge^-$**:** if $T = A_0 \wedge A_1$ and $\mathbb{C}(A_0 \wedge A_1)$`.derivations` is not empty then $\pi_i[A_0 \wedge A_1] \in \mathbb{C}(A_i)$`.derivations` for both $i \in 2$.

(4) $\wedge^+$**:** if $T = A_0 \wedge A_1$ and $\mathbb{C}(A_i)$`.derivations` $\neq \emptyset$ for both $i \in 2$ then $\langle [A_0], [A_1] \rangle \in \mathbb{C}(A_0 \wedge A_1)$`.derivations`.

$\mathbb{C}$ is *ordinally closed with respect to* $\mathcal{T}$ iff for every $T \in \mathcal{T}$ holds:

$$\mathbb{C}(T)\texttt{.depth} = \mathbb{C}(T)\texttt{.desiredDepth}.$$

If both, structurally and ordinally closed with respect to $\mathcal{T}$, we call $\mathbb{C}$ *deductivly closed with respect to* $\mathcal{T}$. Additionally a context is *deductively/structurally/ordinally closed* if and only if it is *deductivly/structurally/ordinally closed with respect to its domain.*

4.4 Properties

Lemma 29. For every finite subset C of constants there exists exactly one deductively closed context $\mathbb{C}$ whose set of free variables is C.

Proof. Existence is obvious. By induction on the depth value of a given context one can achieve the claimed uniqueness. □

Hence the following definition is welldefined:

Definition 30. Let C be a finite set of constants. C^* is the (unique) minimal and deductivly closed context whose set of free variables is C.

Remark. Notice that $\mathrm{FV}(\Box)$ is the invers to $\Box^*$, i.e. $\mathrm{FV}(X^*) = X$ for arbitrary finite sets X of variables. Vice versa $\mathrm{FV}(\mathbb{C})^* = \mathbb{C}$ but only for deductively closed contexts named by $\mathbb{C}$.

So far we tackled deductive closeness. To endorse the decision for the definition of context as done above we consider how answers can be constructed.

Definition 31. Let $\mathbb{C}$ be a deductivly closed context. The set *reachable*$(\mathbb{C})$ is inductively defined by containing

- all those elements of *dom*$(\mathbb{C})$ whose `.depth` value is finite,
- if A and B in *reachable*$(\mathbb{C})$ then $A \wedge B \in$ *reachable*$(\mathbb{C})$ as well, and
- if $B \in$ *reachable*$(\mathbb{C})$ then $A \to B \in$ *reachable*$(\mathbb{C})$ for an arbitrary type A.

Herewith we can show soundness and completeness of the deductivly closed context with respect to the context stable derivations. The first indicates that every type in *reachable*$(\cdot)$ is indeed derivable. The second shows the other way round.

Lemma 32 (Soundness). Let $\mathbb{C}$ be a deductivly closed context. Then for every type T holds

1. if $T \in dom(\mathbb{C})$ and $\mathbb{C}(T)\texttt{.depth}$ is finite then a norm term t exists with $\mathrm{FV}(\mathbb{C}) \vdash_{cs} t : T$ and $\mathrm{depth}(t) = \mathbb{C}(T)\texttt{.depth}$.

2. if $T \in reachable(\mathbb{C})$ then a normal term t exists which satisfies $\mathrm{FV}(\mathbb{C}) \vdash_{cs} t : T$.

Proof.

1. Strong induction on $\mathbb{C}(T)\texttt{.depth}$.

   Base case 1**:** Then there exists a variable $x \in \mathrm{FV}(\mathbb{C})$ of type T.

   Step case $1 \ldots n \to n+1$**:** Assume that $\mathbb{C}(T)\texttt{.depth} = n + 1$. Then there exists an elementary stub term $d \in \mathbb{C}(T)\texttt{.derivations}$ of depth $n+1$. As $\mathrm{depth}_{\mathbb{C}}(S) \leq n$ for every $S \in \mathrm{stubs}(d)$ we can apply the induction hypothesis for S and denote the obtained term as t_S. The demanded term t is d with replaced stubs:

   $$t := d[\ [S] := t_s]_{S \in \mathrm{stubs}(d)}$$

2. By induction on $S \in reachable(\mathbb{C})$. **Induction base**, $\mathbb{C}(S)\texttt{.depth}$ is finite: Use the normalised term of item (1.). **Induction step.** We distinguish between construction rules of $reachable(\mathbb{C})$. For the conjunction case apply pairing to the induction hypotheses. And for the implication use a trivial abstraction on the induction hypothesis.

□

Both items together offer an algorithm to construct a term of a given type if such an term exists. We will detail it in chapter 7. The basic principle is to break the type down and to branch until the domain is hit. For this case the first item offers to traverse parts of the demain in order to pick up derivation stubs and stick them together.

Lemma 33 (Completeness). Let $\mathbb{C}$ be a deductivly closed context. Then for every type T holds

1. if $T \in dom(\mathbb{C})$ and $\mathrm{FV}(\mathbb{C}) \vdash_{cs} t : T$ then $\mathbb{C}(T)\texttt{.depth} \leq \mathrm{depth}(\downarrow t)$. This implies that $\mathbb{C}(T)\texttt{.depth}$ is finite as well.

2. $\mathrm{FV}(\mathbb{C}) \vdash_{cs} t : T$ implies $T \in reachable(\mathbb{C})$ and the existence of a term $t' : T$ built up by stubs, conjuctions and trivial abstractions with $\mathrm{depth}_{\mathbb{C}}(t') \leq \mathrm{depth}(\downarrow t)$ and $\mathrm{stubs}(t') \subseteq \{T \in dom(\mathbb{C}) \mid \mathbb{C}(T)\texttt{.depth} \text{ is finite}\}$.

Proof.

1. Wlog. we can assume that t is in normal form. Strong induction on $\mathrm{depth}(t)$.

   Base case 1**:** Trivial as t is a variable.

   Step case $1, \ldots, n \to n+1$**:** Assume that $\mathrm{depth}(t) = n + 1$. We distinguish t according to its constructional rule. Since they resemble each other only the application rule is considered exemplarily here.
   Thus let $t^T = r^{S \to T} s^S$. Since the depth of both r and s are less or equal n we can apply the induction hypthesis for r and s and obtain:

   $$\begin{aligned} \mathbb{C}(T)\texttt{.depth} &\leq 1 + \max(\mathbb{C}(S \to T)\texttt{.depth}, \mathbb{C}(S)\texttt{.depth}) && \text{(by definition)} \\ &\leq 1 + \max(\mathrm{depth}(r), \mathrm{depth}(s)) && \text{(by induction hypothesis)} \\ &= \mathrm{depth}(t) && \text{(by definition)} \end{aligned}$$

   Notice that the restriction $\vdash_{cs}$ is essential since deductively closeness does not allow proper lambda abstraction.

2. Suppose that t is already normal. Induction on T.
Induction base, T is atomic. Since T is atomic, the outermost term constructor of t cannot be pairing or trivial lambda abstraction. As t is normal its leftmost branch can only consist of applications and projections leading to a free variable at its leaf, say c of type C. According to the typing rules T is a subtype of C. As C is in the domain of $\mathbb{C}$, T is as well and $\mathbb{C}(T)$`.depth` is finite. The latter implies $T \in \mathit{reachable}(\mathbb{C})$. We set t' to $[T]$ and notice that $\text{depth}_{\mathbb{C}}(t') \leq \text{depth}(t)$ because of the first item.
Induction step for T composed. If $T \in \mathit{dom}(\mathbb{C})$ then use $t' = [T]$ as before. Otherwise we claim that if T is built up by implication then t is a trivial lambda abstraction and if T is a conjunction t is a pair. Otherwise we can follow again the leftmost branch with can only consist of applications, projections and variables since t is in normal form. Like before, T needs to be in the domain.
If $T = T_0 \wedge T_1$ set t' to $\langle t_0, t_1 \rangle$ where by t_i is a term offered by the induction hypothesis for the direct subtype T_i of T. Otherwise $T = T_0 \to T_1$ and t' is $\lambda^0 x^{T_0} t_1$ whereby t_1 emerges by the induction hypothesis for T_1. In both cases $T \in \mathit{reachable}(\mathbb{C})$, $\text{depth}_{\mathbb{C}}(t') \leq \text{depth}(\downarrow t)$ and $\text{stubs}(t') \subseteq \{T \in \mathit{dom}(\mathbb{C}) \mid \mathbb{C}(T)$`.depth` is finite$\}$.

□

4.5 Optimality of depth

The mixture of using max during depth calculation and min at `.depth` of every contextitem will pose problems when removing variables later on. Hence, the question whether these definitions are optimal deserves an answer.

4.5.1 Depth on terms

Subsequent chapters prefer the depth value, although the size of terms could have been used without essential modification as well. Even optimality and complexity remain unchanged – at least syntactically, but the cardinalities of the later defined sets $\Delta_{\texttt{struct}}$ and $\Delta_{\texttt{ord}}$ might change. These quantify the contextitems needed to be changed as a result of an `add` or a `remove` operation.

Exactly this is the crucial point, as the choice of the measure function $\Lambda \to \mathbb{N}$ – like the functions $\text{depth}(\cdot)$, $\text{size}(\cdot)$ or whatsoever – indirectly affects the amount of required changes. According to lemma 38 more terms can be covered by depth limitation than by every other measure.

How does it influence the number of updates? Every update of depth attributes for contextitems might enforces an update propagation. An attribute update is the more likely the less terms are covered by the measure function up to the current attribute value. Hence we should be interessed in a measure with a largest possible preimage for every initial segment, to lower the probability for updates.

For the subsequent we omit type annotations and assume implicitly that they were adequate.

Let $f : \Lambda \to \mathbb{N}$ be an opponent of depth-function. Its utilisation requires:

(a) f should ensure finiteness of its argument, i.e. its value is finite if and only if its argument is finite. Infinite terms stem from the coinductive counterpart of Λ. In principle infinite terms could arise by loops. Therefore it is legitimate to require monotonicity, i.e. for every proper subterm s of a finite term t holds $f(s) < f(t)$.

(b) Since terms will be assembled by stubed terms, f should be recursively definable.

(c) Wlog. we can assume that f is independent of the types of the subterms of its argument. Otherwise in the subsequent the functions $f_{\bullet}^{\bullet}$ are to be extended by an additional argument for the type and the sequence η parameterized for every type. However lemma 38 remains unchanged.

All in all we can rewrite f by some auxiliary functions $f_{\bullet}^{\bullet}$ and fix the base cases as follows

$$\begin{aligned} f(x) &= 1 \\ f(\lambda x r) &= 1 + f_1^{abs}(f(r)) \\ f(rs) &= 1 + f_2^{app}(f(r), f(s)) \\ f(\langle r, s\rangle) &= 1 + f_2^{pair}(f(r), f(s)) \\ f(\pi_i r) &= 1 + f_1^{proj_i}(f(r)) && \text{(for } i \in 2) \end{aligned}$$

To avoid gaps in this measure and for technical reasons we require *completeness*, i.e. the existence of a term sequence $(\eta_i)_{i>0}$ such that $f(\eta_i) = i$. Of course one could omit completeness and in return replace $\mathbb{N}$ by the image of f.

Lemma 34.

$$f_1^{\bullet}(x) \geq x$$

for all $x \in \mathbb{N}$ and $\bullet \in \{abs, proj_0, proj_1\}$.

Proof. We detail only the proof for $\bullet = abs$, the others are similar. Let $x \in \mathbb{N}$ be given arbitrarily. Since f is complete there exists an η with $f(\eta) = x$. Hence

$$\begin{aligned} f(\lambda x \eta) &> f(\eta) && \text{(by monotonicity)} \\ &\Rightarrow \\ f_1^{abs}(f(\eta)) &\geq f(\eta) && \text{(by locality of } f) \end{aligned}$$

Since x was arbitrary, $f_1^{abs}(x) \geq x$. □

Lemma 35.

$$f_2^{\bullet}(x, y) \geq max(x, y)$$

for all $x \in \mathbb{N}$ and $\bullet \in \{app, pair\}$.

Proof. Analogous to the proof of lemma 34. □

Lemma 36. For every term t holds depth$(t) \leq f(t)$.

Proof. Induction on t using above lemmata 34 and 35. □

As a consequence a context based on depth value requires no more update operations than a context based on f. Therefore we compare the amount of lambda terms covered up to an arbitrary value in measure of depth and f.

Definition 37 (Covering set). Let $\phi : \Lambda \to \mathbb{N}$ be a function. Its *covering sets* is the function

$$Cov_{\phi}(c) := \{t \in \Lambda \mid \phi(t) \leq c\}$$

from $\mathbb{N}$ to Λ.

For the subsequent let $D(c) := Cov_{\text{depth}}(c)$ and $F(c) := Cov_f(c)$.

Lemma 38. For every $c \in \mathbb{N}$ holds

$$F(c) \subseteq D(c)$$

Proof. Strong induction on c. In its base case $F(0) = 0 = D(0)$. Step case:

$$\begin{aligned} F(c+1) = & Var \\ & \cup\{\lambda xr \mid f_1^{abs}(f(t)) \leq c\} \\ & \cup\{rs \mid f_2^{app}(f(r), f(s)) \leq c\} \\ & \cup\{\langle r, s\rangle \mid f_2^{pair}(f(r), f(s)) \leq c\} \\ & \cup\{\pi_i(r) \mid f_1^{proj_i}(f(t)) \leq c, i \in 2\} \end{aligned}$$

As a difficult case we elaborate on application. The others are treated analogously.

$$\begin{aligned} \{rs \mid f(rs) < c\} &= \{rs \mid f_2^{app}(f(r), f(s)) \leq c\} && \text{(by definiton of } f) \\ &\subseteq \{rs \mid \max(f(r), f(s)) \leq c\} && \text{(by lemma 36)} \\ &= \{rs \mid r \in F(i), s \in F(j), \max(i,j) \leq c\} && \text{(by definition of } F) \\ &\subseteq \{rs \mid r \in D(i), s \in D(j), \max(i,j) \leq c\} && \text{(by induction hypothesis)} \\ &= \{rs \mid \max(\operatorname{depth}(r), \operatorname{depth}(s)) \leq c\} && \text{(by definition of } D) \\ &= \{rs \mid \operatorname{depth}(rs) \leq c+1\} && \text{(by definition of depth)} \end{aligned}$$

All together we obtain

$$\begin{aligned} F(c+1) \subseteq & D(c+1) \\ = & Var \\ & \cup \{\lambda xr \mid \operatorname{depth}(t) \leq c\} \\ & \cup \{rs \mid \max(\operatorname{depth}(r), \operatorname{depth}(s)) \leq c\} \\ & \cup \{\langle r, s\rangle \mid \max(\operatorname{depth}(r), \operatorname{depth}(s)) \leq c\} \\ & \cup \{\pi_i(r) \mid \operatorname{depth}(t) \leq c, i \in 2\} \end{aligned}$$

□

4.5.2 Depth within contexts

Up to now we got acquainted with the depth function as an optimal measure function for our purposes. These values need to be combined per type, say by a function $F : 2^{\overline{\mathbb{N}}} \to \overline{\mathbb{N}}$:

$$\mathbb{C}(T)\texttt{.depth} = F(\{\operatorname{depth}_{\mathbb{C}}(t) \mid t \in \mathbb{C}(T)\texttt{.derivations}\})$$

- F should be easy to compute, i.e. there should be a function $f : \overline{\mathbb{N}}^2 \to \overline{\mathbb{N}}$, such that

  $$F(\emptyset) := \infty$$
  $$F(X \uplus \{c\}) := f(c, F(X))$$

  Hence f is also symmetric.
- The proof of lemma 33 suggests that F, resp. f, should facilitate the choice of a derivation. Hence if $F(X)$ is finite then a value $x \in X$ should exist with $F(X) = x$.
- $\mathbb{C}(T)\texttt{.depth}$ should be finite if and only if a term t exists with $\mathrm{FV}(\mathbb{C}) \vdash_{cs} t$ exists (again inspired by lemma 33).
- F should deter loops during the extraction of terms, i.e. for every context stable derivation l of type T with $T \in \operatorname{stubs}(l)$ should hold

  $$\operatorname{depth}_{\mathbb{C}}(l) > \mathbb{C}(T)\texttt{.depth}.$$

Let $d, L > 1$ be arbitrary. Obviously a deductively closed context $\mathbb{C}$ exists such that

$$|\mathbb{C}(T)\texttt{.derivations}| = 1 \text{ and } \mathbb{C}(T)\texttt{.depth} = d.$$

Let $\mathbb{C}'$ be the deductively closed extentions of $\mathbb{C}$ by

$$c_i : T_i \to T_{i+1} \quad (i \in L)$$

whereby T_0 and T_d denotes T, and T_i is not in the domain of $\mathbb{C}$ for every $0 < i < L$. Observe that now

$$\mathbb{C}'(T)\texttt{.derivations} = \mathbb{C}(T)\texttt{.derivations} \cup \{c_{L-1}[T_{L-1}]\}$$

whereat $\mathbb{C}(T)\texttt{.derivations}$ is treated as an empty set if $T \notin dom(\mathbb{C})$. As $\text{depth}_{\mathbb{C}'}(c_{d-1}[T_{d-1}]) = L + \mathbb{C}'(T)\texttt{.depth}$ we obtain

$$\mathbb{C}'(T)\texttt{.depth} = f(d, \mathbb{C}'(T)\texttt{.depth} + L).$$

Because of the second desideratum f must pick up one of its arguments as its result. The second segregates as the equation

$$L + \mathbb{C}'(T)\texttt{.depth} = \mathbb{C}'(T)\texttt{.depth}$$

can only be solved for $\mathbb{C}'(T)\texttt{.depth} = \infty$. But this contradicts the third requirement. Hence

$$\mathbb{C}'(T)\texttt{.depth} = d = f(d, d + L) = \min(d, d + L)$$

Since d and L ranged over the positive integers f must be min-function. The value of f is dispensable in the case that one of its arguments is 0, since the depth value can never be 0 within a deductively closed context.

All in all, $\mathbb{C}(T)\texttt{.depth}$ <u>must</u> be

$$\min_{t \in \mathbb{C}(T)\texttt{.derivations}} \text{depth}_{\mathbb{C}}(t).$$

4.6 Operational Specification

Under no circumstances we want to rebuild the corresponding deductively closed context after every management operation otherwise operations would cost too much. Requirement three and four on page 5 substantiate the `add` and `remove` operations to:

Definition 39. For every deductively closed context $\mathbb{C}$ and X as a set of variables should hold

1. $\texttt{add}(\mathbb{C}, X) = (\text{FV}(\mathbb{C}) \cup X)^*$, or written as commutative diagram

$$\begin{tikzcd} \square \arrow[r, "\square \cup X"] & \square \arrow[d, "*"] \\ \mathbb{C} \arrow[u, "\text{FV}(\square)"] \arrow[r, "\texttt{add}(\square, X)"] & \square \end{tikzcd}$$

2. $\texttt{remove}(\mathbb{C}, X) = (\text{FV}(\mathbb{C}) \setminus R)^*$, or

$$\begin{tikzcd} \square \arrow[r, "\square \setminus X"] \arrow[d, "\text{FV}(\square)"'] & \square \arrow[d, "*"] \\ \mathbb{C} \arrow[r, "\texttt{remove}(\square, X)"] & \square \end{tikzcd}$$

To implement both algorithms as above specified following lemma is utilised.

Definition 40 ($\sqsubseteq$ on contexts). Let $\mathbb{C}_0$ and $\mathbb{C}_1$ be deductively closed contexts. $\mathbb{C}_0 \sqsubseteq \mathbb{C}_1$ applies if and only if for every type $T \in dom(\mathbb{C}_0)$ holds

1. $dom(\mathbb{C}_0) \subseteq dom(\mathbb{C}_1)$,
2. $\mathbb{C}_0(T)$`.derivations` $\subseteq \mathbb{C}_1(T)$`.derivations`, and
3. $\mathbb{C}_0(T)$`.depth` $\geq \mathbb{C}_1(T)$`.depth`.

Lemma 41. If X and Y are sets of constants with $X \subseteq Y$ then $X^* \sqsubseteq Y^*$

Proof.

1. Let $S \in dom(X^*)$. Then there exists a variable $x : R$ in X such that $T \in \text{subtypes}(R)$. Since c is also in Y, $S \in dom(Y^*)$ as well.
2. Since X^* and Y^* are deductively closed and by use of (1.).
3. Strong induction on $Y^*(\_\_)$`.depth`.

$$\begin{aligned}
Y^*(T).\mathtt{depth} &= \min_{t \in Y^*(T).\mathtt{derivations}} \text{depth}_{Y^*}(t) && \text{(by definition)}\\
&\leq \min_{t \in X^*(T).\mathtt{derivations}} \text{depth}_{Y^*}(t) && \text{(by (2.))}\\
&\leq \min_{t \in X^*(T).\mathtt{derivations}} \text{depth}_{X^*}(t) && \text{(by induction hypothesis)}\\
&= X^*(T).\mathtt{depth} && \text{(by definition)}
\end{aligned}$$

The induction hypothesis is applicable since there exists a term $s \in X^*(T)$`.derivations` such that

$$\text{depth}_{Y^*}(s) = \min_{t \in X^*(T).\mathtt{derivations}} \text{depth}_{Y^*}(t).$$

Thus the depth value in context Y^* of every type S in stubs(s) is strictly less than $depth_{Y^*}(s)$. □

These properties encourage the construction of simple algorithms for both `add` and `remove` since they can rely on a given deductively closed context. Both subsequent chapters detail these management operations.

4.7 Remarks on the implemention of contexts and their items

All relevant datastructures and operations are implemented within the package `context`.

Operations from the outside, notably both management operations (`add`[1] and `remove`, cf. both subsequent chapter) and inquiries (cf. chapter 7), are accepted by the main class `Context`[2]. In the case of management these are are enqueued separately. The invocation of `context.closeDeductively()` commits both queues by further delegation to instances of the class `DeductivelyAdder` and accordingly `DeductivelyRemover`. Thereby we pursue to avoid needless and time consuming multiple updates of the closure.

Every object of class `ContextItem` models an instance of a contextitem as suggested in definition 24. Besides the primary attributes it contains the secondary ones as well.

[1] As the textual outputs are not armed with type for every subterm the implemenation requires that the variable names fix their types. Therefore if a variable a of type A is present in the closure every attempt to add another variable a of a different type will be rejected.

[2] The implementation uses the `Context` class as a singleton, cf [GHJV 95].

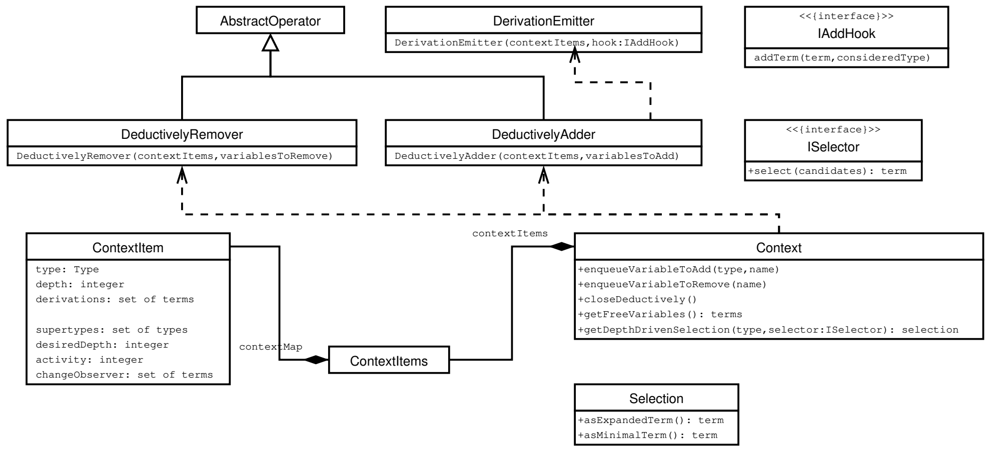


Figure 4.1: An overviewing UML diagram for the package `context`.

Even if secondary attributes are helpful they demand extra work for maintaining. From the point of view of the implementations for `add`, `remove` and query processing the maintaining is hidden by delegation. Therefore the dedicated class `ContextItems` is used on the one hand as a quantified association towards the set of contextitem. On the other hand every operation concerning contextitems are handed over to `ContextItems` which involves the demanded operation of the contextitems and adjust *some* secondary attributes, namely `supertypes`, `activity` and `changeObservers` but not `desiredDepth`. To simplify matters these detours are not mentioned explicitly in the pseudo programming language. In doubt compare to the attached implementation.

Nevertheless `desiredDepth` must be maintained by the core implementation. Consider the repetitive pattern of setting a `depth` value to its `desiredDepth` one. The pseudo code

```
ℂ(T).depth := ℂ(T).desiredDepth;
```

summarizes following (exemplary) Java code:

```
item.commitDepth();

for (Lambda observer: item.getChangeObservers())
{
    ContextItem observerItem = getItem(observer);
    observerItem.updateDesiredDepth(observer);
}
```

These updates have not been encapsulated by the `commitDepth()` methode as the above loop is also used to collect information for further processing.

To hide implementional details from the caller, accesses to attributes are turned into method invocations, for instance `desiredDepth` to `getDesiredDepth()`, as the amount of derivations covered by a certain contextitem is organised as heap.

Akin to conventional traders, our traders are treated to reply primarily on a database storing its data on disks rather than in the main memory, even if the available implementation keeps its information in the main memory. As the input/output bottleneck slows down the trader considerably the runtime is determined by the amount of data base accesses, in our case the access to context items.

During the next three chapters pseudo code treats some variables as sets, whereas they are implemented as heap or as tree and operations on them are not in situ.

For these reasons we focus on the complexity of accesses to contextitems rather on the time complexity.

Chapter 5

Adding Variables

Given a deductively closed context $\mathbb{C}^{\texttt{init}}$ and a set A of variables disjoint from $\mathrm{FV}(\mathbb{C}^{\texttt{init}})$ we want to construct a deductively closed context $\mathbb{C}^{\texttt{final}}$ by an operation called `add` such that $\texttt{add}(\mathbb{C}^{\texttt{init}}, R) = (\mathrm{FV}(\mathbb{C}^{\texttt{init}}) \cup A)^* =: \mathbb{C}^{\texttt{final}}$, or written as commutative diagram (cf. specification 39)

$$\begin{tikzcd}
\square \arrow[r, "\square \cup R"] & \square \arrow[d, "*"] \\
\mathbb{C}^{\texttt{init}} \arrow[u, "\mathrm{FV}(\square)"] \arrow[r, "\texttt{add}(\square, R)"] & \mathbb{C}^{\texttt{final}}
\end{tikzcd}$$

but without rebuilding $\mathbb{C}^{\texttt{final}}$ from its set of free variables. Instead, we want to adjust the given context as lemma 41 proposes.

5.1 Outline of the whole algorithm

At first sight one could be inspired by lemma 41 which treats definition 28 as an inflationary monotone operator and construct $\mathbb{C}^{\texttt{final}}$ as its fixed point by a fixed point iteration based on $\mathbb{C}^{\texttt{init}}$ enriched by the variables of A.

This idea is obviously effective but not necessarily efficient, since indeterminism can lead to an adverse adjustment and vitiate its complexity. To thwart indeterminism it is broken into phases each one basing on scheduling, motivated by following observations.

1. As structural closeness affects only the existence of derivations of the considered type and its direct subtypes, types of low size should be prefered to achieve optimality.

2. Ordinal closeness relies only on the depth value evoked by the existence of derivations. Therefore items are scheduled by their `desiredDepth` values.

As both schedulings strategies are independent the implementation addresses two data structures.

- `pendingTypes` as a minimum heap by type size containing types that could violate structural closeness, initiated as empty set.

- `itemsToGage` as a minimum heap as to `desiredDepth` containing exactly those contextitems whose `desiredDepth` differs from `depth`.

The whole algorithm consists of the following phases:

The 1^{st} phase adds variables of A. In this phase types could be provoked to violate deductive closeness. Those which violate structural closeness are reported in `pendingTypes` and those which violate ordinal closeness in `itemsToGage`.

The 2^{nd} phase closes context structurally based on information in `pendingTypes` and augments `itemsToGage` if necessary.

The 3^{rd} phase closes context ordinally based on information in `itemsToGage`. Afterwards the context is deductively closed.

5.2 Notational issues

At some points the implementation uses contextitems rather than their types, especially in combinations with the java collection framework (see [Sun 04]), to reduce accesses to contextitems by explicit dereferrings like $\mathbb{C}(T)$. To keep proofs and intuitions simple we want to assign an unique context to every step of computation. As bookkeeping operations on contextitems would smear such assignments, types are favoured in the subsequent parts over contextitems. Therefore they are treated as equal and variable names of the implementation are used.

To refer to special contexts we introduce some abbreviations. $\mathbb{C}^{\texttt{init}}$ relates to the initial context and $\mathbb{C}^{\texttt{final}}$ to the final ones. Every phase passes a loop changing possibly the current context. Therefore $\mathbb{C}_i^p$ denotes the context at the beginning of the i^{th} iteration (zero based counted) in phase p. If the i^{th} iteration did not take place it refers to the result of the last iteration, which is explicitly addressed by $\mathbb{C}_\omega^p$. Superscription can be omitted if it is deducible.

These notations outline the interfaces among the phases as:

$\mathbb{C}$	FV	structurally closed w.r.t.	ordinally closed w.r.t.
$\mathbb{C}^{\texttt{init}}, \mathbb{C}_0^1$	$\mathrm{FV}(\mathbb{C}^{\texttt{init}})$	$dom(\mathbb{C})$	$dom(\mathbb{C})$
$\mathbb{C}_\omega^1, \mathbb{C}_0^2$	$\mathrm{FV}(\mathbb{C}^{\texttt{init}}) \cup A$	$dom(\mathbb{C}) \setminus \texttt{pendingTypes}$	$dom(\mathbb{C}) \setminus \texttt{itemsToGage}$
$\mathbb{C}_\omega^2, \mathbb{C}_0^3$	$\mathrm{FV}(\mathbb{C}^{\texttt{init}}) \cup A$	$dom(\mathbb{C})$	$dom(\mathbb{C}) \setminus \texttt{itemsToGage}$
$\mathbb{C}_\omega^3, \mathbb{C}^{\texttt{final}}$	$\mathrm{FV}(\mathbb{C}^{\texttt{init}}) \cup A$	$dom(\mathbb{C})$	$dom(\mathbb{C})$

In the source $\mathbb{C}$ relates to the current context. Implicitly secondary attributes like `supertypes` and `desiredDepth` are updated whenever necessary as well as the maintenance of properties to be a context as declared in definition 28.

To point out complexities we introduce two notations containing those types whose `depth` and respectively `derivations` has been changed:

$$\begin{aligned}
\Delta_{\texttt{ord}} :=& \{T \in dom(\mathbb{C}^{\texttt{init}}) \mid \mathbb{C}^{\texttt{init}}(T).\texttt{depth} \neq \mathbb{C}^{\texttt{init}}(T).\texttt{depth}\} \\
& \cup (dom(\mathbb{C}^{\texttt{final}}) \setminus dom(\mathbb{C}^{\texttt{init}})) \\
\Delta_{\texttt{struct}} :=& \{T \in dom(\mathbb{C}^{\texttt{init}}) \mid \mathbb{C}^{\texttt{init}}(T).\texttt{derivations} \neq \mathbb{C}^{\texttt{final}}(T).\texttt{derivations}\} \\
& \cup (dom(\mathbb{C}^{\texttt{final}}) \setminus dom(\mathbb{C}^{\texttt{init}}))
\end{aligned}$$

For the first two phases an auxiliary method `addTerm`$(t : T)$ is used. It appends an elementary stub term t to $\mathbb{C}(T)$`.derivations`, retains all other primary attributes, maintains all secondaries and preserves the properties to be a context with smallest enhancement of its domain. The latter becomes necessary if T or one of the stubs in t are not in the domain. For details consider the implementation of `addTerm`(t) in `context.ContextItems`.

Lemma 42. `addTerm(t:T)` keeps a context minimal.

Proof. Let $\mathbb{C}$ be the given context and $\mathbb{C}'$ the context after executing `addTerm(t)`. We have to proof in particular $dom(\mathbb{C}') \subseteq \text{subtypes}(\text{FT}(\mathbb{C}'))$. Therefore let $S \in dom(\mathbb{C}')$. If S is already member of $dom(\mathbb{C})$ then also in $\text{FT}(\mathbb{C})$ as $\mathbb{C}$ is presumed to be minimal and such in $\text{subtypes}(\text{FT}(\mathbb{C}'))$. Otherwise $S \in \text{subtypes}(T)$. By construction of $\mathbb{C}'$, $S \in \text{subtypes}(\text{FT}(\mathbb{C}'))$. □

5.3 The Algorithm

5.3.1 First phase (raw adding)

Within this initial phase we append variables in A to the current context $\mathbb{C}^{\texttt{init}}$ and report violations of structural and ordinal closeness to `pendingTypes` and `itemsToGage` respectively.

As all subtypes of some variable in A become part of the domain of the current context, their *seam* to the previous domain might violate deductive closeness (see figure 5.1).

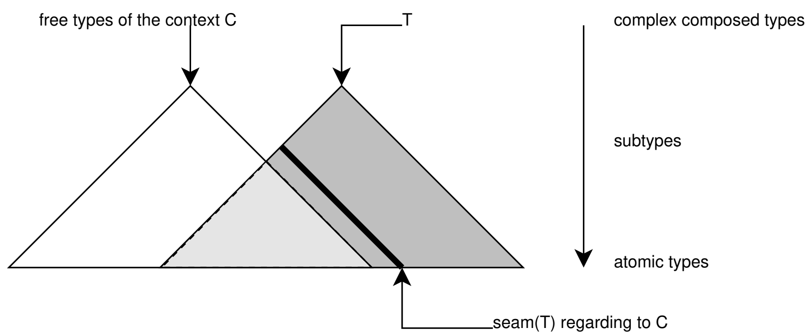


Figure 5.1: Graphical presentation of $seam_{\mathbb{C}}(T)$.

Definition 43. Let $\mathbb{C}$ be a context, $T \in dom(\mathbb{C})$, and $\mathcal{T} \subseteq dom(\mathbb{C})$.

$$seam_{\mathbb{C}}(T) := \{S \mid \text{directSubtypes}(S) \cap dom(\mathbb{C}) \neq \emptyset \text{ and } S \in \text{subtypes}(T) \setminus dom(\mathbb{C})\}$$

Its calculation is covered by the methode `emitSeam` in `context.ContextItems`.

The algorithm for this phase can be paraphrased by

```
pendingTypes := ⋃_{t:T∈A} seam_ℂ(T);
itemsToGage := ∅;

for all t: T ∈ A descending by their size;
{
  call addTerm(t);
  set ℂ(T).depth to 1; // note that its desiredDepth is 1.
  pendingTypes := pendingTypes ∪ {T} ∪ ℂ(T).supertypes;
  for all s of type S in ℂ(T).changeObserver
    if (ℂ(S).desiredDepth ≠ ℂ(S).depth)
      itemsToGage := itemsToGage ∪ {S};
}
```

Next, we show that this algorithm hits its specification.

Lemma 44.

$$\text{FV}(\mathbb{C}^1_\omega) := \text{FV}(\mathbb{C}^{\texttt{init}}) \cup A$$

Proof. Obvious. □

Lemma 45. After executing the first phase `pendingTypes` equals

$$\bigcup_{t:T\in A} (seam_{\mathbb{C}_0}(T) \cup \{T\} \cup \mathbb{C}_\omega(T)\texttt{.supertypes})\,.$$

Proof. Note that due to the descreasing consideration of types the `supertypes` attributes retain for the remaining loop after they had been picked up into `pendingTypes`, as for every type in the domain the type of every element in its `supertypes` field is strictly longer than itself. □

Lemma 46. The context $\mathbb{C}_\omega$ obtained by this phase is structurally closed with respect to $dom(\mathbb{C}_\omega) \setminus$ `pendingTypes`.

Proof. Let $T \in dom(\mathbb{C}_\omega)$. We peruse the properties of structurally closeness one after another.

$\to^+$: Assume that T has shape $A \to B$ and B is derivable within $\mathbb{C}_\omega$. We have to show that $\lambda x^A.[B] \in \mathbb{C}_\omega(A \to B)$`.derivations` or $A \to B \in$ `pendingTypes`. Case $A \to B \in dom(\mathbb{C}_0)$. If B is already derivable within $\mathbb{C}_0$ then $\lambda x^A.[B]$ as well since $\mathbb{C}_0$ is by assumption deductively closed. Otherwise a variable of type B was added and hence $A \to B \in$ `pendingTypes`. Case $A \to B \in dom(\mathbb{C}_\omega) \setminus dom(\mathbb{C}_0)$. Thus a term of type T' with $A \to B \in \text{subtypes}(T')$ must have been added. If $B \in dom(\mathbb{C}_0)$ then $A \to B \in seam_{\mathbb{C}_0}(T')$ and thus also in `pendingTypes`. Otherwise a constant of type B was added and hence $A \to B \in$ `pendingTypes`.

$\to^-$: Assume that T has shape $A \to B$ and both $A \to B$ and A are derivable within $\mathbb{C}_n$. We have to show that $[A \to B][A] \in \mathbb{C}_\omega(B)$`.derivations` or $A \to B \in$ `pendingTypes`. Case $A \to B \in dom(\mathbb{C}_0)$. If a variable of type A or $A \to B$ was added then $A \to B$ is in `pendingTypes`, however. Otherwise $[A \to B][A]$ is in $\mathbb{C}_0(T)$`.derivations` $\subseteq \mathbb{C}_\omega(T)$`.derivations`. Case $A \to B \in dom(\mathbb{C}_\omega) \setminus dom(\mathbb{C}_0)$. Since a variable of type $A \to B$ was added, $A \to B$ is mentioned in `pendingTypes`.

$\wedge^-$: analogous to $\to^-$.

$\wedge^+$: analogous to $\to^+$.

□

Lemma 47. The obtained context $\mathbb{C}_\omega$ is ordinally closed with respect to $dom(\mathbb{C}_\omega) \setminus$ `itemsToGage` and is not ordinally closed with respect to every proper subset.

Proof. Only contextitems whose `.depth` value differs from its `.desiredDepth` value are heaped onto `itemsToGage`. □

Lemma 48. At most $3|\Delta_{\texttt{struct}}|$ accesses to contextitems are required. Those accesses which are only used to dump corresponding types on `itemsToGage` are not connumerated. These will be counted within the succeeding phase.

Proof. An invocation of `addTerm` within a context $\mathbb{C}$ for a term of type T touches $\text{subtypes}(T) \setminus dom(\mathbb{C})$ and their direct subtypes at most once.

As every touched type become part of the domain and every type that at most two direct subtypes the amount of accesses is bounded by $3|\Delta_{\texttt{struct}}|$. □

5.3.2 Second phase (structural closing)

Structual closeness is achieved by following algorithm:

```
while (pendingTypes is not empty)
{
    pick up the top element of pendingTypes as variable type;
    emit derivations for ℂ(type) whereby
      for every term to add "addHook" is invoked;
}
```

We use following auxiliary procedures:

The **emit process** realises definition 28 for a given type. For example $\to^-$ leads to an algorithm working like:

```
// given type labeled by T.
if (T matches A→B)
{
    if (ℂ(A→B).derivations ≠ ∅
       and ℂ(A).derivations ≠ ∅)
        call addTermHook([A→B][A], T);
}
```

For details confer `DerivationEmitter.emitDerivations(ContextItem item)` in package `context`.

addTermHook reports types which could violate structural or ordinal closeness due to appending t to the context.

```
procedure addHook(term t of type T, consideredType)
{
    call addTerm(t);

    if (T had not have any derivation before)                (*)
    {
        dump T on pendingTypes unless T = consideredType;    (**)
        dump  ℂ(T).supertypes on pendingTypes;              (***)
    }

    if (ℂ(T).depth ≠ ℂ(T).desiredDepth)
      add term t to itemsToGage, rsp. update it;
}
```

Lemma 49.

$$\mathrm{FV}(\mathbb{C}^2_\omega) = \mathrm{FV}(\mathbb{C}^2_0)$$

Proof. No variable is added to the context. □

Lemma 50. At the beginning and the end of every loop $\mathbb{C}$ is structurally closed with respect to $dom(\mathbb{C}) \setminus$ `pendingTypes`.

Proof. Initially ensured by lemma 46 and is invariant: By additing terms only its type or its supertypes might be affected. All these were added to `pendingType`. □

Lemma 51. At the beginning and the end of every loop $\mathbb{C}$ is ordinally closed with respect to $dom(\mathbb{C}) \setminus$ `itemsToGage` and is not ordinally closed with respect to every proper subset.

Proof. Initially guaranted by lemma 47 and remains invariant: `addHook` heaps only those contextitems onto `itemsToGage` whose `.depth` values differ from their `.desiredDepth` values. □

Lemma 52. The algorithm of the second phase terminates.

Proof. Since `(*)` can be applied at most once to a type, say S, every type can be heaped at most $1 + |\,\mathrm{directSubtypes}(S)|$ many times because of `(**)` and `(***)`. □

Lemma 53. The set of picked up types is

$$\bigcup_{T\in\Delta_{\mathtt{struct}}} \{T\} \cup \mathbb{C}_0(T)\mathtt{.supertypes} =: B.$$

Proof. Let P the set of those types which had been picked up within the loop.

$P \subseteq B$: Let $T \in P$. We distinguish whether the emit process calls $\mathtt{addTerm}(u : U)$ with a new term u meaning that u does not occur in any `derivations` attribute.

In the positive case $U \in \Delta_{\mathtt{struct}}$. Because of the definition 28 of structural closeness U is either T or subtype of T. Therefore $T \in B$.

Otherwise T has been heaped before because of an appending of a term, say $r : R$. The algorithms of this and the previous phase ensure that T is either R or in $\mathbb{C}(R)\mathtt{.supertypes}$. As $R \in \Delta_{\mathtt{struct}}$, $T \in B$.

$B \subseteq P$: Let $T \in \Delta_{\mathtt{struct}}$. If T is type for a term in A then both T and its supertypes are appended to `pendingTypes` by the first phase and thus considered within the second phase as it terminates.
Otherwise T got derivable due to the emit process applied to T or its supertypes as nobody else can add derivations and $(\mathrm{FV}(\mathbb{C}^{\mathtt{init}})\cup A)^*$ is unique. Since all its supertype are appended to `pendingTypes` at `(**)` and first phase terminates they will be considered later on.

□

Remark. $\Delta_{\mathtt{struct}}$ is unknown in advance. To assess its frontier, i.e.

$$\bigcup_{T\in\Delta_{\mathtt{struct}}} \mathbb{C}_0(T)\mathtt{.supertypes},$$

one has to cross it. Thus the amount of considered types during this second phase can be understood as optimal.

Remark. `pendingTypes` is organised as a minimum heap according to the type size to minimize the amount of accesses to contextitems, as the emit procedure relies only at the given type and its direct subtypes.

Lemma 54. The amount of contextitem accesses is bounded by $3|B|$ as defined in the above lemma.

Proof. By lemma 53 at most the contextitems for types in B are considered and by the proof of lemma 52 each is considered at most three times with the same argumentation as in the proof of lemma 48. □

5.3.3 Third phase (ordinal closing)

Ordinal closeness is achieved by following algorithm:

```
for (ContextItem item: itemsToGage)
{
    item.desiredDepth := item.depth;                              (*)

    for all s of type S in item.changeObserver
        if (ℂ(S).desiredDepth ≠ ℂ(S).depth)
            itemsToGage.add(S);
}
```

Lemma 55.

$$\mathrm{FV}(\mathbb{C}^3_\omega) = \mathrm{FV}(\mathbb{C}^3_0)$$

Proof. This phase does not touch any `derivations` attributes. □

Lemma 56. For every i and all types T in `itemsToGage` holds:

$$\mathbb{C}_i(T)\texttt{.desiredDepth} < \mathbb{C}_i(T)\texttt{.depth}$$

Proof. Initially ensured by the previous phase and remained by ascertaining whether depth differs from desired depth before adding a type to `itemsToGage`. □

Lemma 57. At the beginning and the end of every loop $\mathbb{C}$ is ordinally closed with respect to $dom(\mathbb{C}) \setminus \texttt{itemsToGage}$ and is not ordinally closed with respect to every proper subset.

Proof. Initially guaranted by lemma 51 and remains invariant: Only those contextitems are heaped onto `itemsToGage` whose `.depth` values differ from their `.desiredDepth` values. □

Corollary 58. The `depth` values can only lower or remain.

Proof. By above lemma. □

Lemma 59. The sequence of the `desiredDepth` field of considered types is weakly increasing.

Proof. It suffices to show that the minimum of the `desiredDepth` values of all types mentioned in `itemsToGage` does not decrease.

Let T be the type picked up. Lemma 56 and line `(*)` decreases the `depth` value of $\mathbb{C}(T)$. Thus at most the `desiredDepth` value changes of those types for which a term in $\mathbb{C}(T)$`.changeObservers` of its type exists. The `desiredDepth` value can thereby only remain or decrease, but can not fall below the new $\mathbb{C}(T)\texttt{.depth} + 1$ value by this commit.

Therefore the minimum of `itemsToGage` can not increase. □

Lemma 60. Every type in $\Delta_{\texttt{ord}}$ was considered exactly once and every other in $dom(\mathbb{C}^{\texttt{final}})$ never.

Proof. Firstly every type in $\Delta_{\texttt{ord}}$ was considered but none of the complement with respect to $dom(\mathbb{C}^{\texttt{final}})$. Thus it remains to show that every type in $\Delta_{\texttt{ord}}$ was considered at most once. For contradiction assume that $T \in \Delta_{\texttt{ord}}$ was treated twice. Let d_1 its desired depth at the first time and d_2 at the second.

By lemma 59 $d_1 \leq d_2$ holds. Since T can not occur twice at the same time, T was heaped a second time <u>after</u> T had been considered the first time. Again by lemma 59 and the definition of `desiredDepth`, d_2 must be strictly greater than d_1.

This means that a type $S \in \mathrm{stubs}(t)$ for some $t \in \mathbb{C}(T)$`.derivations` exists whose depth value had been strictly increased, which contradicts lemma 58. □

Thus the algorithm of the third phase is optimal for the amount of contextitem accesses and leads directly to its terminations.

5.3.4 Complexity

All in all the number of accesses during an invocation of `add` is bounded by

$$\theta\left(|\Delta_{\texttt{ord}}| + \left|\bigcup_{T\in\Delta_{\texttt{struct}}} \{T\} \cup \mathbb{C}^{\texttt{init}}(T)\texttt{.supertypes}\right|\right).$$

Or in other words: nearly linear in the amount of necessary updates.

If we presume a thin and approximately uniformly distributed closure the amount of supertypes is bounded by a constant. Ergo, the whole amount of access is linear in the amount of mandatory updates.

Chapter 6

Removing Variables

Given a deductively closed context $\mathbb{C}^{\texttt{init}}$ and a subset R of $\mathrm{FV}(\mathbb{C}^{\texttt{init}})$ we want to construct a deductively closed context $\mathbb{C}^{\texttt{final}}$ by an operation called `remove` such that $\texttt{remove}(\mathbb{C}^{\texttt{init}}, R) = (\mathrm{FV}(\mathbb{C}^{\texttt{init}}) \setminus R)^* =: \mathbb{C}^{\texttt{final}}$, or written as commutative diagram (cf. specification 39)

$$\begin{tikzcd} \Box \arrow[r, "\Box\setminus R"] & \Box \arrow[d, "*"] \\ \mathbb{C}^{\texttt{init}} \arrow[u, "\mathrm{FV}(\Box)"] \arrow[r, "\texttt{remove}(\Box{,}R)"] & \mathbb{C}^{\texttt{final}} \end{tikzcd}$$

but without rebuilding $\mathbb{C}^{\texttt{final}}$ from its set of free variables. Instead, we want to adjust the given context as proposed by lemma 41.

We use the same ornaments for contexts as proposed in section 5.2 on page 38.

6.1 Outline of the whole algorithm

As the algorithm has to assure both minimality (cf. definition 25) and deductively closeness we decompose it into two phases.

The 1^{st} **phase** dislodges all constants in R from their corresponding `.derivations` and constrict the domain. This leads of course to a not necessarily ordinally closed but minimal and structurally closed context.

The 2^{nd} **phase** adjusts the `depth` values to obtain a deductively closed context $(\mathrm{FV}(\mathbb{C}^{\texttt{init}}) \setminus R)^*$.

Swapping both parts does not accelerate this combination as minimalisation does not benefit from deductive closeness. But the other way around is beneficial: As the main complexity is based on the second part (as we will see later), the preferred minimalisation leads to a better overall efficiency.

During this chapter we discuss only a corresponding pseudo algorithm to restrain administrative circumstantiality. For details refer to the implementing class `context.DeductivelyRemover`.

We introduce following abbreviations about context changes

$$\begin{aligned} \Delta_{\texttt{ord}} &:= \{T \in dom(\mathbb{C}^{\texttt{final}}) \mid \mathbb{C}^{\texttt{init}}(T)\texttt{.depth} \neq \mathbb{C}^{\texttt{final}}(T)\texttt{.depth}\} \\ \Delta_{\texttt{struct}} &:= dom(\mathbb{C}^{\texttt{init}}) \setminus dom(\mathbb{C}^{\texttt{final}}) \end{aligned}$$

to state complexity issues more precisely.

6.2 The Algorithm

6.2.1 First phase (raw removal)

6.2.1.1 Specification

To argue more accurately about the removal of variables and the minimalisation of the domain we introduce some notations.

Definition 61. Let $\mathbb{C}$ be a context and r a constant ot type R. Their difference $\mathbb{C}-r$ is a context such that

- $dom(\mathbb{C}-r) := dom(\mathbb{C}) \setminus \{T \in \text{subtypes}(R) \mid \mathbb{C}(T).\texttt{activity} = 1\}$,
- $\text{FV}(\mathbb{C}-r) := \text{FV}(\mathbb{C}) \setminus \{r\}$, and
- for all $T \in dom(\mathbb{C}-r)$ holds

$$\begin{aligned}(\mathbb{C}-r)(T).\texttt{depth} :=& \mathbb{C}(T).\texttt{depth} \text{ and}\\ (\mathbb{C}-r)(T).\texttt{derivations} :=& \{t \in \mathbb{C}(T).\texttt{derivations}\\ &\mid \text{stubs}(t) \subseteq dom(\mathbb{C}-r)\} \setminus \{r\}\end{aligned}$$

The only precarious challenge for well definedness is its domain. The definition of the `activity` attribute relaxes to

$$\begin{aligned}dom(\mathbb{C}\ -\ r : R) =& dom(\mathbb{C}) \setminus \{T \in \text{subtypes}(R) \mid \mathbb{C}(T).\texttt{activity} = 1\}\\ =& \bigcup_{a:A\in\text{FV}(\mathbb{C})} \text{subtypes}(A)\\ & \setminus \{T \in \text{subtypes}(R) \mid \mathbb{C}(T).\texttt{activity} = 1\}\\ =& \bigcup_{a:A\in\text{FV}(\mathbb{C})} \text{subtypes}(A)\\ & \setminus \{T \in \text{subtypes}(R) \mid \forall a : A \in \text{FV}(\mathbb{C}) \setminus \{r\}. T \notin \text{subtypes}(A)\}\\ =& \bigcup_{a:A\in\text{FV}(\mathbb{C})\setminus\{r\}} \text{subtypes}(A)\\ =& \bigcup_{a:A\in\text{FV}(\mathbb{C}-r)} \text{subtypes}(A)\end{aligned}$$

Definition 62. Let $\mathbb{C}$ be a context and R a finite set of a constants. Their difference $\mathbb{C}-R$ is defined by induction on the cardinality of R to

$$\begin{aligned}\mathbb{C} - \emptyset &:= \mathbb{C}\\ \mathbb{C} - (R \uplus \{r\}) &:= (\mathbb{C} - R) - r\end{aligned}$$

As the perambulation of R does not effect the result this difference is well defined.

Lemma 63. Let $\mathbb{C}$ be a deductively closed context and R a set of variables. Then $\mathbb{C}-R$ is a structurally closed context and for all $T \in dom(\mathbb{C}-R)$ holds

$$(FV(\mathbb{C}) \setminus R)^*(T).\texttt{derivations} \ = \ (\mathbb{C}-R)(T).\texttt{derivations}$$

and

$$(FV(\mathbb{C}) \setminus R)^*(T).\texttt{depth} \ \geq \ (\mathbb{C}-R)(T).\texttt{depth}$$

Proof. Abbreviate $(FV(\mathbb{C}) \setminus R)^*$ as A.

- Let $t \in (\mathbb{C} - R)$`.derivations` $\subseteq \mathbb{C}$`.derivations`. As $dom(\mathbb{C} - R) = dom(A)$ and A is deductively closed, $t \in A$`.derivations`. And the other way around:

$$
\begin{aligned}
& A(T)\texttt{.derivations} & \\
=& \{t \in A(T)\texttt{.derivations} \mid \text{stubs}(t) \subseteq dom(A)\} & \text{(by definition of context)} \\
\subseteq& \{t \in \mathbb{C}(T)\texttt{.derivations} \mid \text{stubs}(t) \subseteq dom(\mathbb{C} - R)\} & \text{(by lemma 41)} \\
=& (\mathbb{C} - R)(T)\texttt{.derivations} & \text{(by definition of } \mathbb{C} - R\text{)}
\end{aligned}
$$

- The `depth` value of $(\mathbb{C} - R)(T)$ equals that of $\mathbb{C}(T)$. Lemma 41 yields the inequality.

□

Thus, after removal of variables only an adjustment of the `depth` values is required to obtain a minimal and deductively closed context $\mathbb{C}^{\texttt{final}}$.

6.2.1.2 Implementation

The class `context.DeductivelyRemover` delegates the minimalisation to `context.contextItems.removeTerms`. The crux is to detect whether an item survives or not. Therefore an additional administrative attribute `activity` was introduced to contextitems as reference counter as mentioned in section 4.3 on page 26. It allows the following procedure:

> If the term to remove is a variable, decrease the `activity` value of all subtypes. If the value zero is hit, remove the corresponding contextitem exhaustive, by meaning to remove all derivations of its type and all derivations using stubs of its type.

There the amount of contextitems access is bounded linearly by $\Delta_{\texttt{struct}}$.

This phase results in a minimum heap `itemsToGage` ordered by the `.desiredDepth` attribute consisting of exactly those elements whose `.desiredDepth` values differ from `.depth`.

6.2.2 Second phase (depth adjustment)

During this final phase only the `depth` values need to be adjusted as above lemma 63 suggests. All other structures like domain, active types, set of free variables, derivations etc. remains unchanged.

6.2.2.1 Intractability

At first sight one could proceed akin to the third phase of the `add` algorithm with following adaptions:

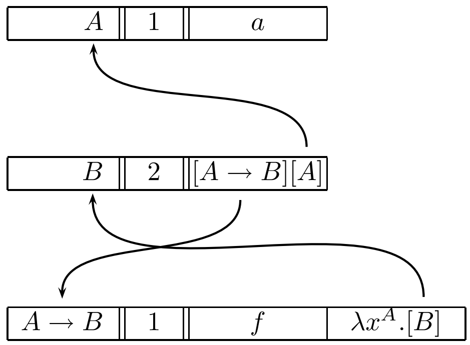
```
while (a type T ∈ dom(ℂ) exists such that
  ℂ(T).depth<ℂ(T).desiredDepth)
{
    choose such a T;
    ℂ(T).depth := ℂ(T).desiredDepth;
}
```

Obviously this leads to the desired context, if it terminates. And this is the crucial point, *if it terminates.*

For example consider the deductive closure for

$$\begin{aligned} a:&\quad A\\ f:&\quad A\to B\\ \mathit{cover}:&\quad (A\to B)\to X \end{aligned}$$

Here *cover* is only used to avoid effects of minimalisation and is suppressed as it becomes irrelevant for the subsequent. Its context could be illustrated as



whereby the middle value is the `depth` value.

The observation of its <u>unique</u> trace for removing f reveals the reason for non termination:

time	$\mathbb{C}(A)$.`depth`	$\mathbb{C}(B)$.`depth`	$\mathbb{C}(A\to B)$.`depth`
0	1	2	1
1	1	2	3
2	1	4	3
3	1	4	5
4	1	6	5
$\vdots$	$\vdots$	$\vdots$	$\vdots$
∞	1	∞	∞

Thus a refinement of the above raw algorithm is needed to avoid such infinite loops.

6.2.2.2 Depth boundedness

A loophole could be tendered by the observation that every depth value is bounded by the cardinality of the current domain or it is infinite, as it can be proven by induction on the depth value.

This leads to teh following refinement by replacing the above loop body by:

```
choose such a T;
if (ℂ(T).desiredDepth > |dom(ℂ)|) then
    ℂ(T).depth := ∞;
else
    ℂ(T).depth := ℂ(T).desiredDepth;
```

If $\mathbb{C}(T)$.`depth` exceeds $|dom(\mathbb{C})|$ it becomes sooner or later infinite, more precisely for some limit ordinals which describe the progress of computation. If the depth is set to infinity it will not be considered any more since $\infty < \alpha$ is wrong for every $\alpha \in \overline{\mathbb{N}}$. Termination follows immediately as every type in the domain is considered at most $|dom(\mathbb{C})|$-many times.

So far we achieved an algorithm, but its complexity is not really viable. Assume that a variable should be removed such that the amount of touched types is negligible compared to the number of active types. For instance consider once again the previous example but enriched by a multitude of variables whose types do not overlap with $A \to B$ (and hence also not with A and B). Since the removal of f leads to a loop wich is to be iterated according to above algorithm approximately $|dom(\mathbb{C})|/2$ times, this procedure is not contenting. In particular we stipulate that the complexity for removing depends only on the number of changed contextitems and not on the whole context.

Thus we have to look for a further refinement which can handle loops accurately.

6.2.2.3 Refinement of the second phase

As pointed out in the previous chapter a heap `itemToGage` is used which contains precisely those contextitems which violate ordinal closeness and is used to set `depth` to `desiredDepth` if it is considered for the first time. Therefore an additional bookkeeping set `visited` contains all previously considered contextitem within the current `remove` invocation. If an item is repeatedly visited `depth` values are adjusted only within `visited` and items outside `visited`, which need thereby to update their depth, are dumped on `itemsToGage`. Thus the loop problem is shifted to the adjustment within `visited`: the depth values of dependent items within `visited` are set to infinity – the neutral element of min – and afterwards rearranged as per the third phases of the `add` algorithm.

Definition 64. Let $\mathbb{C}$ be a context and $T \in dom(\mathbb{C})$.
$dependents(T)$ is the smalltest set of active types with

> if S is member or equals T then the types of all terms in
> $\mathbb{C}(S)$`.changeObservers` are members as well.

Remark. Even if $dependents(\cdot)$ depends to the context and should formally be written as $dependents_{\mathbb{C}}(\cdot)$ we omit this context annotation as during this phase structural components remain constant as well as the domain.

Definition 65. Let $\mathbb{C}$ be a context and $\mathcal{T}$ a subset of $dom(\mathbb{C})$.

$$\begin{aligned} border(\mathcal{T}) &:= \text{type}\left(\bigcup_{T\in\mathcal{T}} \mathbb{C}(T)\texttt{.changeObservers}\right) \setminus \mathcal{T} \\ bordered(\mathcal{T}) &:= \mathcal{T} \cup border(\mathcal{T}) \end{aligned}$$

Before we turn to the main part of the algorithm we introduce two auxiliary methods, `infinite-Depth` and `adjustDepth`. The intended invocation is `adjustDepth(infiniteDepth(`T_0`))`. Let $\mathbb{C}$ denote the context immediately before evaluating `infiniteDepth`, $\mathbb{C}'$ after `infiniteDepth` and before `adjustDepth`, and $\mathbb{C}''$ after all. Analogously for `itemsToGage`. Both auxiliary methods keep `visited` unchanged.

6.2.2.4 Method `infiniteDepth`

The method `infiniteDepth` with argument T_0 as type changes the current context $\mathbb{C}$ to $\mathbb{C}'$ with

$$\mathbb{C}'(S)\texttt{.depth} = \begin{cases} \infty & \text{if } S \in dependents(T_0) \cap \texttt{visited} \\ \mathbb{C}(S)\texttt{.depth} & \text{otherwise} \end{cases}$$

whereby every other attribute remains stable and returns the set of possible changed types, $dependents(T_0) \cap$ `visited`, whereby `visited` refers to the variable of the main loop. The content of `itemsToGage` remains untouched. Thus we obtain

Lemma 66. If $\mathbb{C}$ is ordinally closed with respect to a set $\mathcal{T}$ of types then $\mathbb{C}'$ is ordinally closed with respect to

$$\mathcal{T} \setminus \mathit{bordered}(\mathit{dependents}(T_0) \cap \texttt{visited}).$$

Proof. Let Z abbreviate $\mathit{dependents}(T_0) \cap \texttt{visited}$ and let $T \in \mathcal{T} \setminus \mathit{bordered}(Z)$. As $Z \subseteq \mathit{bordered}(Z)$ the `depth` attribute of type T remains. So, it is left to show $\mathbb{C}'(T)\texttt{.desiredDepth} = \mathbb{C}(T)\texttt{.desiredDepth}$. By the definitions of this attribute and of $\mathrm{depth}_{\mathbb{C}}(\cdot)$ it suffices to proof that $\mathbb{C}(S)\texttt{.depth} = \mathbb{C}'(S)\texttt{.depth}$ for every term $t \in \mathbb{C}(T)\texttt{.derivations}$ and every type $S \in \mathrm{stubs}(t)$. Therefore assume the opposite. Then $\mathbb{C}(S)\texttt{.depth}$ was set of infinity and hence $S \in Z$. The definition of `changeObserver` ensures $t \in \mathbb{C}(S)\texttt{.changeObserver}$. These result in

$$T \in \mathrm{type}\left(\bigcup_{R \in Z} \mathbb{C}(R)\texttt{.changeObserver}\right) \subseteq \mathit{bordered}(Z).$$

Contradiction to $T \in \mathcal{T} \setminus \mathit{bordered}(Z)$. □

As its implementation bases on a depth first search restricted on `visited` its complexity is linear in the cardinality of

$$\bigcup_{T \in \mathit{dependents}(T_0) \cap \mathit{bordered}(\texttt{visited})} \mathbb{C}(T)\texttt{.derivations}$$

and requires at most

$$|\mathit{dependents}(T_0) \cap \texttt{visited}|$$

accesses to contextitems.

6.2.2.5 Method `adjustDepth`

The method `adjustDepth` expects a set $\mathcal{T}$ of types equaling $\mathit{dependents}(T_0) \cap \texttt{visited}$ for some $T_0 \in \mathit{dom}(\mathbb{C})$ whose `depth` attributes are infinite.

In return it adjusts the current context $\mathbb{C}'$ to $\mathbb{C}''$ with

$$\mathbb{C}''(S)\texttt{.depth} = \begin{cases} \mathbb{C}(S)\texttt{.desiredDepth} & \text{if } S \in \mathcal{T} \\ \mathbb{C}(S)\texttt{.depth} & \text{otherwise} \end{cases},$$

leaves all other attributes untouched and amplifies `itemsToGage` properly.

```
// input 𝒯 as a set of types as mentioned in the text.

Declare itemsToAdjust as minimum heap according to desiredDepth
  initialised with 𝒯 restricted to those whose
  desiredDepth attribute differs from its depth;

while (itemsToAdjust ≠ ∅)
{
    pick up the topmost element of itemsToAdjust as type T;
    if (T ∉ visited)
        heap T on itemsToGage;                              (*)
    else
    {
        ℂ(T).depth := ℂ.desiredDepth;
        for every s:S ∈ ℂ(T).changeObservers
            if (ℂ(S).depth ≠ ℂ(S).desiredDepth)
                heap S on itemsToAdjust;                    (**)
    }
}
```

Let Δ`itemsToGage` denote the set of all types added to variable `itemsToGage` due to the line marked by `(*)`. Hence $\texttt{itemsToGage}'' = \texttt{itemsToGage} \cup \Delta\texttt{itemsToGage}$.

Lemma 67 (properties of `itemsToAdjust`). `itemsToAdjust` is a subset of *dependents*(T_0) and for of its elements T holds:

- $\mathbb{C}(T)\texttt{.depth} \neq \mathbb{C}(T)\texttt{.desiredDepth}$, and
- if $T \notin \texttt{visited}$ then $T \in \mathit{border}(\mathit{dependents}(T_0) \cap \texttt{visited})$

Proof. Initially warranted by the first `if` statement. Its perpetuation is to be verified at point `(**)`. By definition of *depenents*(T_0) and by previous incarnation of this invariant, S as mentioned in the source is an element of it. Further S is only heaped if its `depth` differs from `desiredDepth`. Let additionally $S \notin \texttt{visited}$. By invariant $\texttt{T} \in \texttt{visited} \cap \mathit{dependents}(T_0)$ and thus S lies in

$$\text{type}\left(\bigcup_{T \in \texttt{visited} \cap \mathit{dependents}(T_0)} \mathbb{C}(T)\texttt{.changeObservers}\right).$$

But as $S \notin \texttt{visited} \cap \mathit{dependents}(T_0)$, S is in $\mathit{border}(\mathit{dependents}(T_0) \cap \texttt{visited})$. $\square$

Lemma 68. Δ`itemsToGage` is a subset of

$$\{T \in \mathit{border}(\mathit{dependents}(T_0) \cap \texttt{visited}) \mid \mathbb{C}(T)\texttt{.depth} \neq \mathbb{C}(T)\texttt{.desiredDepth}\}$$

Proof. Consider the line marked by `(*)` and use the last item of above lemma 67. $\square$

As this algorithm resembles the third phase of the `add` algorithm restricted to `visited`, argumentations about soundness, completness and complexity issues apply accordingly.

Lemma 69. If $\mathbb{C}$ is ordinally closed with respect to $\mathcal{C}$ then $\mathbb{C}''$ is ordinally closed with respect to

$$\mathcal{C} \setminus \Delta\texttt{itemsToGage}.$$

Proof. According to lemma 66, $\mathbb{C}'$ is ordinally closed with respect to

$$\mathcal{C} \setminus \mathit{borders}(\mathit{dependents}(T_0) \cap \texttt{visited}).$$

As this algorithm recycles the third phase of the `add` method but with a main loop restricted to `visited`, the `depth` attribute for all types in *dependents*$(T_0) \cap$ `visited` are adjusted and all other types on its border and violating ordinal closeness are covered by Δ`itemsToGage` (lemma 68). $\square$

6.2.2.6 Main part

```
while (itemsToGage ≠ ∅)
{
    pick up the topmost element of itemsToGage as T;
    if (T ∈ visited)
        adjustDepth(infiniteDepth(T));
    else
    {
        visited := visited ∪ {T};
        ℂ(T).depth := ℂ(T).desiredDepth;
        if (thereby the depth value changed)
            for every type S of some term ℂ(T).changeObservers
                if (ℂ(S).depth ≠ ℂ(S).desiredDepth)
                    dump S on itemsToGage;
    }
}
```

Lemma 70 (Termination). The main loop is passed at most $|\Delta_{\texttt{ord}}|^2$ times.

Proof. It suffices to show that the measure

$$(|\Delta_{\texttt{ord}}| - |\texttt{visited}|, |\texttt{itemsToGage} \cap \texttt{visited}|)$$

decreases according to its lexicographical order because both `visited` and `itemsToGage` are subsets of $\Delta_{\texttt{ord}}$ at all times.

Hence let T be a picked up type. Assume it is in `visited`. As both `adjustDepth` and `infiniteDepth` do not change the set `visited`, it remains constant. Since T was picked up out of `itemsToGage` it decreases and both above mentioned methods do not heap any visited type, `itemsToGage` $\cap$ `visited` decreases.

Next assume $T \notin$ `visited`. As hence `visited` increases by one, the measure decreases. □

Lemma 71 (Soundness). The main loop maintains the invariant that the current context $\mathbb{C}$ is ordinally closed with respect to $dom(\mathbb{C}) \setminus$ `itemsToGage`.

Proof. Initially ensured by previous phase. For the retaining of the invariant during the loop let T be a type picked up.
If $T \in$ `visited` lemma 69 ensures the claimed ordinal closeness.
Otherwise ($T \notin$ `visited`) T was removed from `itemsToGage` and its `depth` value was equaled to `desiredDepth`. Every type whose `depth` value differs thereby from `desiredDepth` was dumped on `itemsToGage`. □

As this phase only changes the `depth` value and the given context is minimal and structurally closed the context afterwards is minimal and deductively closed and equals $(\mathrm{FV}(\mathbb{C}) \setminus R)^*$ as demanded.

Lemma 72. The second phase accesses $\mathcal{O}(|\Delta_{\texttt{ord}}|^3)$ to contextitems.

Proof. Both auxiliary methods access the items a linear amount of time. The amount of their invocations is quadratically bounded due to lemma 70. □

6.2.3 Complexity

The total amount of accesses to contextitems for `remove` is bounded by

$$\theta(|\Delta_{\texttt{ord}}|^3 + |\Delta_{\texttt{struct}}|)$$

due to subsection 6.2.1.2 and lemma 72.

The non-wellordering $(\overline{\mathbb{N}}, \geq)$ hamper termination for a naive adjustment of `depth` values as demonstrated in chapter 6.2.2.1. Therefore the compensating bookkeeping set `visited` is used. Both the non-wellordering and the bookkeeping could hold responsible for the occurrence of an additional factor $|\Delta_{\texttt{ord}}|$ in comparison with the previous management operation `add`.

We do not detail the runtime complexity of the whole algorithm as many adminstrational operations are hidden in the implementation not presented here, even if they are logarithmically or linearly bounded.

Chapter 7

Queries

7.1 Selection

So far we discussed deductivly closed contexts and elementary management functions on it. Now we are confronted with the task to answer queries for a certain type by derivations of this type. For this task mainly the lemmata 32 and 33 will be utilised.

7.1.1 Necessity of Selection

Even if lemma 32 points out how to construct proper derivations the first item of this lemma leaves at some points open how to choose an elementary stub term. Thus the multitude of possible answers can grow exponentially measured in the amount of involved types whose contextitems cover more than one elementary stub term of minimal depth. Hence selections are essential.

Lemma 73. There exists a sequence of finite sets Γ_n and types A_n such that the amount of normal derivations d_n with $\Gamma_n \vdash d_n : A_n$ is exponential in the size of Γ_n.

Proof. For every $n \geq 0$ choose Γ_n as the set containing for each $i \in n$:

$$\begin{array}{rccc@{\qquad}rccc} fork_{i,0}\colon & B_i & \to & A_{i+1} & fork_{i,1}\colon & C_i & \to & A_{i+1} \\ join_{i,0}\colon & A_i & \to & B_i & join_{i,1}\colon & A_i & \to & C_i \end{array}$$

and as initial part

$$init : A_0.$$

Following context stable terms (recall the definition on page 21) of type A_n are in normal form whose free variables are covered by Γ_n:

$$\begin{array}{l} fork_{n-1,\beta(n-1)}(join_{n-1,\beta(n-1)}(\\ \qquad \ldots \\ \qquad fork_{1,\beta(1)}(join_{1,\beta(1)}(\\ \qquad\qquad fork_{0,\beta(0)}(join_{0,\beta(0)} init) \\ \qquad)) \\ \qquad \ldots \\ )) \end{array}$$

for every function $\beta : n \to 2$. Hence we obtained at least 2^n normal forms. □

Assume a context X^* with $X \supseteq \Gamma_n$ for some n and Γ_n as in the previous proof. If a user inquires a derivation for type A_n we need to design how the system should answer in order to explore the multitude of derivations.

We discuss some answer patterns:

One answer is choosen indeterminately. As the system should act as a database system it needs to comply with the ACID principle (Atomicity, Consistency, Isolation, and Durability), cf. [KeEi 99, section 9.5]. But fortuitousness weakly violates *isolation* since it pretends the existence of another user: suppose a user inquires the same question twice and the answers differ he could infer the existence of another user who has removed some essential part of the former derivation in the meantime.

Arbitrary, but fixed all along. But how should the user react if the proffered answer does not suffice? For example if he distrusts one of the mentioned service providers within the derivation. Precisely nothing, as the system would not offer any possibility to vary the answer for a fixed requested type.

Offer all possible derivations explicitly. Well, this solves obviously the above problem but may be exponential – as shown above – and is hence not really practicable.

All possible derivations given as cursor. Indeed the cursor concept [KeEi 99, section 4.20] could revise the above unapproachability. Hereby the user obtains an interface to traverse all possible derivations. Its benefit is that the user decides within the first few terms whether his intended answer occurs or not. A modified principle was put into practice by many web search engines, as a complete list of all websites containing given words would not help the user in general. This modification consists of a representation block by block containing a bounded amount of results. A grave drawback is caused by locking the involved context. Update operations can not take place until all users have released their cursor interface.

Preselection As indicated the user knows in advance how the enquired answer should look like to fulfill his demands. To achieve this we allow the user to hand over

- a selection strategy (cf. subsequent section 7.3) declaring which parameters of the terms should be optimised, and
- a selection can advise the trader to choose a proper elementary stub term by delegating the choice back to the enquirer and help the enquirer to get an answer sufficing her demands.

Therefore, we favour preselection as advertised in the introducing chapter 1 by the activity diagram on page 7.

7.1.2 Definition

Definition 74 (Selection). Let $\mathbb{C}$ be a deductivly closed context. A *selection* is a pair

$$(S, \quad \sigma : \mathit{reachable}(\mathbb{C}) \hookrightarrow \Lambda_{\text{stub}})$$

such that

- $S \in \mathit{reachable}(\mathbb{C})$
- for every $T \in \mathit{reachable}(\mathbb{C})$ holds

$$\sigma(T) \begin{cases} \in \mathbb{C}(T)\texttt{.derivations} & \text{if } T \in \mathit{dom}(\mathbb{C}) \\ = \langle [T_0], [T_1] \rangle & \text{if } T \notin \mathit{dom}(\mathbb{C}) \text{ and } T = T_0 \wedge T_1 \\ = \lambda^0 x^{T_0}[T_1] & \text{if } T \notin \mathit{dom}(\mathbb{C}) \text{ and } T = T_0 \to T_1 \end{cases}$$

- σ is *closed* concerning $\mathbb{C}$, i.e. for all $T \in dom(\sigma)$ and $S \in \text{stubs}(\sigma(T))$ also $S \in dom(\sigma)$ holds.
- σ is *acyclic* concerning $\mathbb{C}$, i.e. there exists an injective map $\eta : dom(\sigma) \to |dom(\sigma)|$ such that for all $T \in dom(\sigma)$ and $S \in \text{stubs}(\sigma(T))$ holds $\eta(T) < \eta(S)$.
- σ is *minimal*, i.e. every proper subset of σ is not closed.

The last bullet implies $\eta(S) = 0$. We refer to S as the *type* of the above selection.

Remark. The second item reproduces the structure of $reachable(\cdot)$ as defined on page 27. For elements of the domain of the context the derivations can be choosen among all terms mentioned in their contextitem. Otherwise the derivations are fixed.

Remark. The witnessing function η is only needed for $[\![(S,\sigma)]\!]_{\min}$ (see below). Whenever this representation is demanded η is constructed implicitly within `Selection.asMinimalTerm` in package `context` since σ can be treated as an acyclic graph and η is a congruent topological order used to traverse the selection. (in doubt confer [OtWi 96, section 8.1]).

Lemma 75. For every deductively closed context $\mathbb{C}$ holds:
$\text{FV}(\mathbb{C}) \vdash^{?} T$ if and only if a selection for $\mathbb{C}$ of type T exists.

Proof. By lemma 32 and 33. □

Definition 76 (Free Variables and Size of a Selection). Let (S,σ) be a selection. Its set of free variables is

$$\text{FV}((S,\sigma)) := \bigcup_{t \in range(\sigma)} \text{FV}(t)$$

And its size is the cardinality of $dom(\sigma)$.

7.2 Representations as λ terms

7.2.1 As expanded terms

First, we consider a simple representation of selections as expanded lambda term.

Definition 77 (As expanded term). Let (S,σ) be a selection.

$$[\![(S,\sigma)]\!]_{\text{expanded}} := \sigma(S)[\,[T] := [\![(T,\sigma)]\!]_{\text{expanded}}]_{T \in \text{stubs}(\sigma(S))}$$

Notice that this definition is well defined: Acyclicity ensures termination, closeness ensures that (T,σ) is a selection as well, and as $[\![\cdot]\!]_{\text{expanded}}$ does not contain stubs, the order of stubs substitution does not effect its value.

Lemma 78. $[\![(S,\sigma)]\!]_{\text{expanded}}$ is in normal form for every selection (S,σ).

Proof. The only possibility to violate normal form is due to redexes. For contradiction assume that a redex $(\lambda x^A r^B)s$ occurs in t. Both redex and its body r need to have idential type B. This contradicts the acyclicity of the selection since

$$\begin{aligned} &A \to B \in \text{stubs}(\sigma(B)) && \text{(because of the application)} \\ &B \in \text{stubs}(\sigma(A \to B)) && \text{(due to the lambda abstraction)} \end{aligned}$$

□

In full first order minimal logic the gap between normal and minimal forms is known to be hyperexponential provided by Orevkov*'s sequence* (cf. [TrSc 96, section 6.7.6]).

We show that although we only consider a fragment of the simple typed lambda calculus which correlates with the propositional minimal logic by the CURRY-HOWARD *isomorphism*, the corresponding gap is exponential.

Lemma 79. A sequence $\mathbb{C}_n$ of deductively closed contexts together with a sequence of selection (P_n, σ_n) exists such that

$$\theta(|dom(\sigma_n)|) = \mathrm{FV}(\mathbb{C}_n)$$

but $|[\![(P_n, \sigma_n)]\!]_{\mathrm{expanded}}|$ is exponential in $\mathrm{FV}(\mathbb{C}_n)$.

Proof. Let

$$\mathbb{C}_n := (\{pair_0 : P_0\} \cup \{pair_i : P_i \wedge P_i \to P_{i+1} \mid i < n\})^* \,.$$

For type P_n exists exactly one selection (P_n, σ_n) with

$$\begin{aligned}\sigma_n := &\{(P_0, & pair_0)\} \cup & \\ &\{(P_{i+1}, & [P_i \wedge P_i \to P_{i+1}][P_i \wedge P_i]) & \mid i \in n\} \cup \\ &\{(P_i \wedge P_i \to P_{i+1}, & pair_i) & \mid i \in n\} \cup \\ &\{(P_i \wedge P_i, & \langle [P_i], [P_i] \rangle) & \mid i \in n\}\end{aligned}$$

Thus $|[\![(P_n, \sigma_n)]\!]_{\mathrm{expanded}}| = 2^{n+2} - 3 =: s_n$ as it satisfies

$$\begin{aligned} s_0 &= 1 \\ s_{i+1} &= 2s_i + 3 && \text{(for } i \in n) \end{aligned}$$

whereby the value 3 stems from the variable, application, and pairing per every type P_i for $1 < i \leq n$. ∎

7.2.2 As minimal terms

As the straightforward representation of selections as expanded terms can cause an exponential growth measured by the size of their domains, an alternative is necessary but at the price of a somewhat more complex algorithm.

Definition 80 (Degree). Let (S, σ) be a selection.
For every type $T \in dom(\sigma)$ its degree $deg(T)$ is the number of occurences of $[T]$ within the range of σ. For example $[T]$ occures in $\langle [T], [T] \rangle$ twice.

Definition 81 (As minimal term). Let (S, σ) be a selection.

$$[\![(S, \sigma)]\!]_{\min} := [\![(S, \sigma)]\!]_{\min}^{|dom(\sigma)|-1}$$

for every $0 < i < |dom(\sigma)|$

$$[\![(S, \sigma)]\!]_{\min}^{i} := \begin{cases} (\lambda x.[\![(S, \sigma)]\!]_{\min}^{i-1} \, [\, [T_i] := x])t_i & \text{if } deg(T_i) > 1 \text{ (and } x \text{ fresh)} \\ \quad [\![(S, \sigma)]\!]_{\min}^{i-1} \, [\, [T_i] := t_i] & \text{if } deg(T_i) = 1 \end{cases}$$

and

$$[\![(S, \sigma)]\!]_{\min}^{0} := \sigma(S) \qquad (= t_0)$$

whereby $t_i = \sigma(T_i)$ and $T_i = \eta^{-1}(i)$. Note that η is injective, cf. definition 74.

Lemma 82. Let (S, σ) be a selection.

$$[\![(S, \sigma)]\!]_{\min} \to_{\beta}^{*} [\![(S, \sigma)]\!]_{\mathrm{expanded}}$$

Proof. Execute all redexes produces by the $[\![\, \cdot \,]\!]_{\min}^{\cdot}$ functions from outside inwardly. ∎

Next, we show that $[\![\,\cdot\,]\!]_{\min}$ produces a term whose size is linear (with a low hidden factor) in the minimal size among all β-equal terms.

Lemma 83. Let (S, σ) be a selection.

$$|dom(\sigma)| \quad \leq \quad |[\![(S,\sigma)]\!]_{\min}| \quad \leq \quad 5|dom(\sigma)|$$

Proof. The first inequality arises from the manifestation of every type within the domain of σ as it retains the type of its argument and σ is minimal.
As a preparation for the proof of the last inequality it can be easily seen by induction on i that for every $T \in dom(\sigma)$ the amount of occurences of $[T]$ in $[\![(S,\sigma)]\!]^i_{\min}$ is equal to the amount among $t_0, \dots, t_i$.
Recall that for every $T_i \in dom(\sigma)$, $[T_i]$ can only be a subterm of $\sigma(T_j)$ if $j < i$ since η induces a topological order.
To achieve the inequality we assign costs of at most 5 to every type of the domain of η for their contribution to the size of $|[\![(S,\sigma)]\!]_{\min}|$. Thereby the whole term is covered completely.
For $i > 0$ the substitution $[\,[T_i] := x]$ at the upper case does not influence the length of $[\![(S,\sigma)]\!]^{i-1}_{\min}$ as $|[T_i]| = |x|$. Thus its contribution to the whole size is $2 + |t_i| \leq 5$, sharply.
At the lower case the substitution $[T_i := x]$ increases the length of $[\![(S,\sigma)]\!]^{i-1}_{\min}$ by at most $|t_i| - 1 \leq 5$ as $[T_i]$ occurs exactly once within $[\![(S,\sigma)]\!]^{i-1}_{\min}$.
For $i = 0$ the contribution is $|t_0| \leq 5$. □

Lemma 84. Let (S, σ) be a selection.

$$|[\![(S,\sigma)]\!]_{\min}| \leq 5 \min \left\{ |t| \;\middle|\; t \to^*_\beta [\![(S,\sigma)]\!]_{\text{expanded}} \right\}$$

Proof. As every beta reduction step reduces the diversity of types at most by the type of the lambda abstraction belonging to the redex to be executed, t contains every type mentioned in $range(\sigma)$, because σ is additionally minimal. Thus

$$\begin{aligned} |[\![(S,\sigma)]\!]_{\min}| &\leq 5|dom(\sigma)| && \text{(by lemma 83)} \\ &\leq 5|t| && (\sigma \text{ preserves the argument's type}) \end{aligned}$$

for arbitrary terms t normalising to $[\![(S,\sigma)]\!]_{\text{expanded}}$. □

7.2.3 Comparison to system F

The reason why we are actually not interested in terms of exact minimal length among all β equivalent terms is simple: systems more expressive than the simple lambda calculus might have shorter minimal forms by additional rules which allow to pack information more efficiently.

An example is of course *system F* (cf. [Matt 00, chapter 7], [Matt 04, chapter 2.1]), a lambda calculus equipped with parametric polymorphism like used in SML or in Java 1.5.

We just recall the essential extensions:

- dedicated typevariables as special types, say $X, Y, \dots$, and an additional rule for type construction: if X is a typevariable and T a type, $\forall X.T$ is a type as well. The concept of free typevariable and type substitution are meant as usually.
- two term constructor rules:

  $$\frac{r : R}{\Lambda X.r : \forall X.R} \qquad \text{(type abstraction)}$$

  Hereby the so called *variable condition* must be satisfied, i.e. X is not free in any free variable of r. For example $\Lambda X \lambda x^X x : \forall X.X \to X$ fulfills this condition but $\Lambda X x^X$ does not.
  To instantiate a type abstraction one can use

  $$\frac{r : \forall X.R}{rS : R[X := S]} \qquad \text{(type appliction)}$$

- a reduction rule:
$$(\Lambda Y r^R)X \to_\beta r^{R[Y:=X]}$$

We define a set Γ_n as

$$c_{i,k} : \bigwedge_{j\in i} A_{j,k} \to A_{i,k} \qquad \text{(for } i,k \in n\text{)}$$
$$f : \bigwedge_{i\in n} A_{n-1,i} \to X$$

to point out the difference in sense of complexity between the simple typed lambda calculus and system F.

An implication whose premise is a vacuous meta conjunction (i.e. $\bigwedge$) is treated as its conclusion and meta conjunctions are treated right associative, for example:

$$\bigwedge_{j\in 3} A_{j,k} = A_{0,k} \wedge (A_{1,k} \wedge A_{2,k})$$

Lemma 85. Every derivation t of type X with $\mathrm{FV}(t) \subseteq \Gamma_n$ within the simple typed lambda calculus has length $\Omega(n^3)$.

Proof. The normal form of t contains of at least the following types

$$\bigwedge_{l\leq j\leq r} A_{j,k}$$

for every $k \in n$ and $0 \leq l \leq r < n$. As these $\theta(n^3)$ types must occur also in t its size is bounded by $\theta(n^3)$.
Hereby it is essential that $\bigwedge$ produces rightwise associated conjunction: the expansion of $\bigwedge$ is defined in opposite direction in comparison to its associativity. If leftwise were choosen instead only $\theta(n^2)$ many types would occur in the normal form of t. □

Lemma 86. A derivation t of type X with $FV(t) \subseteq \Gamma_n$ within system F whose length is $\mathcal{O}(n^2)$ exists.

Proof. Consider following arrangement similar to that above

$$\begin{array}{rl} c_0 : & A_0 \\ c_1 : & A_0 \to A_1 \\ c_2 : & A_0 \wedge A_1 \to A_2 \\ \vdots & \quad \ddots \\ c_{n-1} : & A_0 \wedge (A_1 \wedge (\cdots A_{n-2})) \to A_{n-1} \end{array}$$

As a selection (A_{n-1}, σ) for the context $(\{c_i \mid i \in n\})^*$ with $|dom(\sigma)|$ quadratic in n exists, $[\![(A_{n-1}, \sigma)]\!]_{\min} =: \mu$ is quadratic in n as well (by lemma 83).

Second order abstraction manages to achieve the desired derivation of type X

$$\begin{array}{l} (\lambda a. \\ \quad f \\ \qquad (a \ \ A_{0,0} \cdots A_{n-1,0} \ \ c_{0,0} \cdots c_{n-1,0}) \\ \qquad \vdots \\ \qquad (a \ \ A_{0,n-1} \cdots A_{n-1,n-1} \ \ c_{0,n-1} \cdots c_{n-1,n-1}) \\ ) \\ \quad (\Lambda A_0 \ldots \Lambda A_{n-1} \lambda c_0^{A_0} \ldots \lambda c_{n-1}^{A_0 \wedge (A_1 \wedge (\cdots A_{n-2})) \to A_{n-1}} \mu) \end{array}$$

of length $\theta(n^2)$. Hereby the complexity is independent of whether the lengths of the annotated types mentioned in type abstraction or application are counted or not. □

Hence, representations in system F are more efficient than in the simple typed lambda calculus.

Selection and $[\![\cdot]\!]_{\min}$ are to be extended in the usual way for system F. A corresponding lemma to lemma 83 holds as well.

Although we can obtain in the general case shorter terms, polymorphism is not implemented in our program since the advantage is small due to two observations. On one hand the occurrence of structurally equivalent patterns for a sufficiently long path is fairly unlikely.

On the other hand the construction of selections becomes more expensive. The time to construct a selection might be quadratic in the size of the selection. Therefore we use the observation that verification is a lower bound for construction.

Assume a context $\mathbb{C}$, a type A in its domain, and a selection (σ, A) for this type. Let $\mathbb{C}'$ be the above context multiplied $|dom(\sigma)| =: k$ times by copying type and variable structures and let A be spread over $A_1, \ldots, A_k$. The size of the induced selection for $A_1 \wedge \ldots \wedge A_k$ in system F is bounded by $\theta(k)$ due to construction. On the other hand, the selection for system F bases on a modification of an appropriate selection for simple typed terms. The determination of its pairs which can be merged by type abstraction costs $\theta(k^2)$. Every pair requires $\theta(k)$ comparisons and k groups exist containing exactly those parts which can be merged.

7.2.4 Comparison to CoCoGraphs

Another kind of representation is offered by the *CoCoGraph* mentioned in [Krau 03, in particular section 5.3] for the first time. It turns out that CoCoGraphs are essentially lambda terms:

CoCoGraph	lambda term
node	n-ary application to a variable
wrapper node	β redex
repeated usage of an output port	β redex
edge	symbol for construction order
input port	part of an application sequence
output port	part of a conjunction
(not presentable!)	abstraction
(not presentable!)	differentiation between sequential application and application to conjunctions

For instance the CoCograph of figure 7.1 (taken from [Krau 03, figure 5.9]) can be converted into the equivalent lambda term

$$(\lambda x \langle \mathtt{CTB}(\mathtt{CTA}\ \mathtt{s1})(\pi_0 x), (\pi_1 x)\rangle)(\mathtt{OJA}\ \mathtt{s2})$$

or for better comparison to the CoCoGraph the term can be considered as a directed acyclic graph. Note that the double usage of `OJA` needs to be translated into a β redex and the input into the variables `s1` and `s2`.

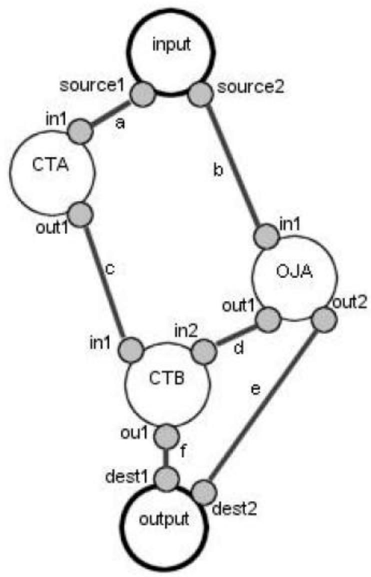
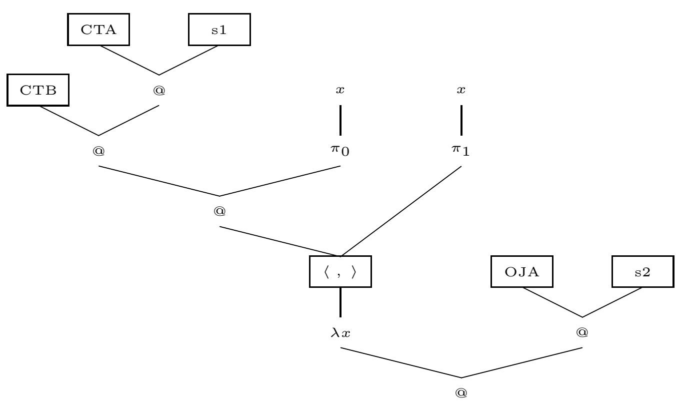


Figure 7.1: An example of an CoCoGraph.



(@ denotes applications.)

Thus like lambda terms CoCoGraphs are not optimal for minimal representation of work flows to compute additional context information as demonstrated in the preceding subsection.

7.3 Selection Strategies

Given a deductively closed context and a type enquired by the user we want to discuss two reasonable possibilities to gain an adequate selection.

7.3.1 Depth Driven Selection

Since contextitems are already provided with `depth` values we use a depth-driven approach for the construction of selections.

Definition 87. Let $\mathbb{C}$ be a deductively closed context and $S \in dom(\mathbb{C})$. A *depth driven selection* (S, σ) is any selection which satisfy for every T in the domain of $\mathbb{C}$:

$$\text{depth}_{\mathbb{C}}(\sigma(T)) = \mathbb{C}(T)\texttt{.depth} < \infty.$$

As the choice of the terms is left open a depth driven selection does not need to be unique. Furthermore the above definition is well defined as σ is acyclic: Assume for contradiction that σ is cyclic. Then a sequence $(T_i)_{i \in k}$ of length k exists with $T_i \in dom(\sigma)$ for all $i \in k$ and:

$$T_{(i+1)\, mod\, k} \in \text{stubs}(\sigma(T_i)) \qquad \text{(for all } i \in k)$$

This implies that all types of the loop are in the domain of the context: Assume the converse. Note that the domain of the context is closed under subtypes by its definition. Hence if T_i is not in this domain $T_{(i-1)\, mod\, k}$ is not as well. Iteration ensures that every type of the cycle is not in the domain. By definition of selection the type size decreases strictly for cyclic increasing indices. This contradicts the finiteness of type size.

The definition of ordinally closeness enforces

$$\mathbb{C}(T_i)\texttt{.depth} \geq 1 + \mathbb{C}(T_{(i+1)\, mod\, k})\texttt{.depth} \qquad \text{(for all } i \in k)$$

meaning

$$\mathbb{C}(T_0)\texttt{.depth} > \mathbb{C}(T_1)\texttt{.depth} > \ldots > \mathbb{C}(T_{k-1})\texttt{.depth} > \mathbb{C}(T_0)\texttt{.depth}$$

as all `depth` values have to be finite, which leads to an obious falsity.

`context.Context.getDepthDrivenSelection` implements the construction of a depth driven selection. It breaks down the inquired type constructors, according to the definition 31 of $reachable_{\mathbb{C}}(\cdot)$, until it hits a element of the domain of the context whose depth is finite. Thenceforward it picks up successively derivation stubs whose depth equals the depth of its contextitem. For simplification a first fit selection is used by the implementation, but could be easily be adapted for other purposes.

7.3.2 Free Variables Minimal Selection

As the multitude of content providers might be held responsible for "contamination" of the quality of information due to error propagation, we should also be interested in derivations involving a minimum amount of free variables.

Given a deductively closed context $\mathbb{C}$ and a type T we are looking for an algorithm constructing a selection (T, σ) with a minimal amount of free variables provided that such an selection exists at all.

Starting from an arbitrary selection for a type T, we stepwise approach the optimum by a binary search. As it already suffices for the most purposes to approximate the minimal amount of free variables the parameter `granularity` allows to specify how much the cardinality of the set of free variables of the result may differ from its minimum.

During each loop the function `SBS(T, s)` tests whether a selection of type T with at most s free variables exists. If so, a corresponding selection is returned. We detail the functionality of this auxiliary parameter later.

```
// Input: type T; granularity as natural number, default is 0.
// Enviroment: context.

Let (T, τ) be an arbitrary selection for T
  for example "depth driven";
up := size of [[(T, τ)]]min;
down := 1;

while (up-down > granularity)
{
  middle := (down+up) / 2;

  if (SBS(T, middle) returns successfully)
    up := middle;
  else
    down := middle+1;
}

return with SBS(T, down);
```

The critical point is SBS as its task is **NP**-complete.

For the sequencing lemma we recall short the language SAT (cf.[Reis 90, section 6.3]). Given a propositional formula φ build by a conjunction of clauses. A clause is a disjunction of literals – either a variable, denoted by $\{x_i\}_{i\in n}$, or its negation –, saying

$$\varphi = \bigwedge_{i\in k}\bigvee_{j\in 3} x_{var(i,j)}^{pol(i,j)}$$

where $var : k \times 3 \to n$, $pol : k \times 3 \to 2$ and $x^0 = \neg x$ as well as $x^1 = x$. $\bigvee$ and $\bigwedge$ are to be treaten as meta symbols meaning a finite enrolling to $\vee$ and $\wedge$ on the language level. The question is whether this formula is satisfiable according to the standard interpretation, i.e. whether an assignment $\beta : n \to 2$ exists such that $\beta \models F$. This problem is the established example of **NP** complete languages.

Definition 88 (Problem of Size Bounded Selection). The *problem of size bounded selection* (SBS for short) is to decide for a given context $\mathbb{C}$, a type T, and an integer value s whether a selection (T, σ) exists with $|\operatorname{FV}(\sigma)| \leq s$.

Lemma 89. SBS is **NP**-complete.

Proof. SBS $\in$ **NP.** Guess a (raw) selection and verify whether it is wellformed and its amount of free variables is bounded by s.

NP $\leq_p$ SBS. It suffices that $3 - \text{SAT}$ reduces to the problem above.

Let a formula φ be given in conjunctive normal form with exactly three literals per clause

$$\bigwedge_{i\in k}\bigvee_{j\in 3} x_{var(i,j)}^{pol(i,j)}$$

whereby $var : k \times 3 \to n$ and $pol : k \times 3 \to 2$.

This formula is translated in polynominal time into the deductive closure of

$$\begin{aligned}
val_{i,p} :& X_{i,p} \\
var_{i,p} :& X_{i,p} \to X_i && \text{(for } i \in n \text{ and } p \in 2) \\
clause_{i,j} :& X_{var(i,j),pol(i,j)} \to C_i && \text{(for } i \in k \text{ and } j \in 3) \\
force :& \bigwedge_{i \in n} X_i \wedge \bigwedge_{i \in k} C_i \to G
\end{aligned}$$

with G as goal and $2n + k + 1$ as threshold s.

Apparently G is derivable by a selection containing all above mentioned constants.

It remains to show that this transformation respects the border of both languages: φ is satisfyable if and only if there exists a derivation whose set of free variables has size bounded to s:

$\Rightarrow$ Let $\beta : \{x_i\}_{i \in n} \to 2$ (0 is interpreted as false) be a satisfying assignment to φ. Then there exists a selection with free variables $force$, $clause_{i,j}$ ($i \in k$ and some j with $\beta(x_{var(i,j)}) = pol(i,j)$, which must exists as β satisfies every clause), as well as $val_{i,\beta(i)}$ and $var_{i,\beta(i)}$ $(i \in n)$.

$\Leftarrow$ Let (G, σ) be a selection whose set of free variables is bounded by $2n+k+1$. The variable $force$ as well as $clause_{i,j}$ for every $i \in k$ and some $j \in 3$ must be in $\mathrm{FV}((G, \sigma))$. Hence at most $2n$ variables have to be distributed among $\{val_{i,p}, var_{i,p} \mid i \in n, p \in 2\}$. According to the pigeonhole principle the exists a function $\beta : \{x_i\}_{i \in n} \to 2$ such that both $val_{i,\beta(i)}$ and $var_{i,\beta(i)}$ are elements of $\mathrm{FV}((G, \sigma))$. By construction of the $clause_{i,j}$s β is a satisfying assignment for φ.

□

Even if SBS can be translated into a SATquestion and could be delegated to an SAT*solver*[1] as oracle using its result to form a proper selection, we don not favour it since its implementation as a rather huge part of the context needs to be embedded into the SATquestion. A further parameter bounding the depth of the demanded selection might be useful to get a grip on the run time of both, translation and orcale.

To avoid multiple invocations of a SATorcale it is imaginable to run the translation once and then to ask for a model with a minimal amount of variables assigned to true among those variables encoding a free variable of the translated part of the context.

Anyway, it is open to be implemented.

[1] Note that many rather efficient SATsolvers, in the tenor of low run time for small instances, are available like exemplarily `sato` (available at `http://www.cs.uiowa.edu/~hzhang/sato.html`) or `zchaff` (available at `http://www.princeton.edu/~chaff/software.html`)

Chapter 8

Examples

To introduce the implementation of a deductive trader, called Ganglia, some raw setting are itemised below. A ganglion is the kernel of a nerve cell[1]. The implemention's name is choosen as stubed terms look like dendrites, both contextitems and nerve cells own each time generally more than one, and stubs accord with axons. Since a deductively closed context consists of a multitude of contextitems the noun "ganglion" is pluralised.

To keep the implementation simple the selection strategy is fixed to "depth driven" and the selector to "first fit". Both can be replaced by more appropriate ones.

8.1 New in town?

Assume someone is moving to another town. To get an accommodation the landlord claims a bank account and that he should apply personally. On the other hand the local branch of his home bank wants him to hand in his identity card to open an account. To get such a card he should go to the local registration office.

To achieve the indicated procedure we encode the involved actors as functions and inquire for a minimal lambda term describing such an procedure.

```
 --   --         --         --
|    |  | |\  | |    |   | |  |
| -  |__| | \ | | -  |   | |__|
|__| |  | |  \| |__| |__ | |  |

Enter a command, as example "help" for help.
ganglia> add identity: Identity
[ OK ] "identity" was queued to adding.
ganglia> add registrationOffice: Identity -> IdentityCard
[ OK ] "registrationOffice" was queued to adding.
ganglia> add bank: IdentityCard * Identity -> BankAccount
[ OK ] "bank" was queued to adding.
ganglia> add landlord: Identity -> BankAccount -> Flat
[ OK ] "landlord" was queued to adding.
ganglia> dc
======== statistics ========
 Time elapsed ...
  ... to order terms and determinate borders:  0,080 seconds,
```

[1]see http://en.wikipedia.org/wiki/Ganglion

```
   thereby 4 terms were considered.
  ... to close context structurally:           0,060 seconds,
   thereby 20 items were considered.
  ... to close context ordinally:              0,000 seconds,
   thereby 5 items were considered.
---------------------------
 Total time elapsed for adding deductively:    0,140 seconds.
===========================
ganglia> min Flat
Derivation is:
  landlord identity (bank ((registrationOffice identity),identity)): Flat

Its depth is 5 and
its size 11.
ganglia> sml "examples/town.sml"
ganglia>
```

Ganglia outputs a "playground" for SML in the file `town.sml`:

```
(* Section models datatypes of every reasonable atomic type *)
datatype Identity = valueOfIdentity;
datatype Flat = valueOfFlat;
datatype IdentityCard = valueOfIdentityCard;
datatype BankAccount = valueOfBankAccount;


(* Section models free variable as function returning only test values *)
val identity: Identity =
  valueIdentity;
val bank: IdentityCard*Identity->BankAccount =
  fn _:(IdentityCard*Identity) => valueOfBankAccount;
val landlord: Identity->BankAccount->Flat =
  fn _:(Identity) => fn _:(BankAccount) => valueOfFlat;
val registrationOffice: Identity->IdentityCard =
  fn _:(Identity) => valueOfIdentityCard;
```

After loading the above code and entering the derivation produced by Ganglia, SML outputs:

```
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 [CM&CMB]
- use "examples/town.sml";
[opening examples/town.sml]
datatype Identity = valueOfIdentity
datatype Flat = valueOfFlat
datatype IdentityCard = valueOfIdentityCard
datatype BankAccount = valueOfBankAccount
val identity = valueOfIdentity : Identity
val bank = fn : IdentityCard * Identity -> BankAccount
val landlord = fn : Identity -> BankAccount -> Flat
val registrationOffice = fn : Identity -> IdentityCard
val it = () : unit
- landlord identity (bank ((registrationOffice identity),identity));
val it = valueOfFlat : Flat
-
```

So far we can only conclude that SML accepts the typing and computes the expected value. As SML evaluates in applicative order [Bry 04, subsection 3.2.2] the output for `min Flat` can be detailed as follows:

`landlord identity (bank ((registrationOffice identity),identity))`
$\rightarrow_\beta^*$ `landlord valueOfIdentity (bank`
`((registrationOffice valueOfIdentity),valueOfIdentity))`
$\rightarrow_\beta^*$ `landlord valueOfIdentity (bank`
`(valueOfIdentityCard,valueOfIdentity))`
$\rightarrow_\beta^*$ `landlord valueOfIdentity (valueOfBankAccount)`
$\rightarrow_\beta^*$ `valueOfFlat`

The first reduction steps replaces those variables by their definitions whose types are atomic. The subsequent steps execute those redexes produced by replacing variables by their definitions in applicative order.

The reduction sequence coincides with the intuition. It can be summarised by the following trace:

interstation	things you should at least take with you
	`valueOfIdentity`
`registrationOffice`	
	`valueOfIdentity, valueOfIdentityCard`
`bank`	
	`valueOfIdentity, valueOfBankAccount`
`landlord`	
	`valueOfFlat`

whereby "things you should at least take with you" are represented by the free variables. "`valueOfIdentity`" denotes yourself.

8.2 What does drive most other provers mad?

We consider the context of the proof to lemma 79 for $n = 260$ and inquire type P_{260}. As demonstrated there, the minimal term is linear in n whereas its normal form is linear in 2^n. Latter means that a prover plainly using normal forms would need memory for approximately 2^{260} intermediate step to decide its derivability. This would exceed the amount of atoms in our univers, approximately 10^{78} in number.

```
 --   --         --         --
|    |  | |\  | |    |   | |  |
| -  |__| | \ | | -  |   | |__|
|__| |  | |  \| |__| |__ | |  |

Enter a command, as example "help" for help.
ganglia> pair 260
ganglia> dc
======== statistics ========
 Time elapsed ...
  ... to order terms and determinate borders:  0,751 seconds,
   thereby 261 terms were considered.
  ... to close context structurally:           1,052 seconds,
   thereby 1817 items were considered.
  ... to close context ordinally:              0,080 seconds,
   thereby 520 items were considered.
----------------------------
 Total time elapsed for adding deductively:    1,883 seconds.
============================
```

```
ganglia> min P260
Derivation is:
  (fn v0 => (fn v1 => (fn v2 => (fn v3 => (fn v4 =>
[...]
(pair_4 <v2,v2>)) (pair_3 <v1,v1>)) (pair_2 <v0,v0>)) (pair_1 <pair_0,pair_0>): P260

Its depth is 521 and
its size 1818.
ganglia>
```

Every prover whose derivation search involves, even if only implicitly, the construction of the full normal form would not terminate in this universe.

8.3 Removal

We consider as an example the intractability instance discussed in subsection 6.2.2.1. In practice a removing could be required for instance due to a outage of content providers because of a network error or whatsoever, in particular for longer periods.

```
 --   --         --         --
|    |  | |\  | |    |   | |  |
| -  |__| | \ | | -  |   | |__|
|__| |  | |  \| |__| |__ | |  |

Enter a command, as example "help" for help.
ganglia> add a:A
[ OK ] "a" was queued to adding.
ganglia> add f:A -> B
[ OK ] "f" was queued to adding.
ganglia> add cover:(A -> B) -> X
[ OK ] "cover" was queued to adding.
ganglia> dc
======== statistics ========
 Time elapsed ...
  ... to order terms and determinate borders:  0,050 seconds,
   thereby 3 terms were considered.
  ... to close context structurally:           0,040 seconds,
   thereby 11 items were considered.
  ... to close context ordinally:              0,000 seconds,
   thereby 2 items were considered.
----------------------------
 Total time elapsed for adding deductively:    0,090 seconds.
============================
ganglia> show
context:
Number of elements: 11
+-----------+----------+--------------+---------------------------------------------+...+
| type      | depth    | desiredDepth | derivations                                 |   |
+-----------+----------+--------------+---------------------------------------------+...+
| A         | 1        | 1            | [a: A]                                      |   |
| B         | 2        | 2            | [([A->B]) ([A]): B]                         |   |
| X         | 2        | 2            | [([(A->B)->X]) ([A->B]): X]                 |   |
| A->B      | 1        | 1            | [f: A->B, fn v0 => [B]: A->B]               |   |
| (A->B)->X | 1        | 1            | [fn v0 => [X]: (A->B)->X, cover: (A->B)->X] |   |
+-----------+----------+--------------+---------------------------------------------+...+
```

```
free variables: [a: A, f: A->B, cover: (A->B)->X]
scheduled constants to add: []
scheduled constants to remove: []

ganglia> remove f
[ OK ] "f" was queued to removing.
ganglia> dc
======== statistics ========
 Time elapsed ...
  ... to order reduce to minimal context:    0,000 seconds,
   thereby 1 terms were considered.
  ... to close context ordinally:            0,100 seconds,
   thereby 4 items were considered,
   whereby 3 items repeatedly.
   1 times items were considered repeatedly.
----------------------------
 Total time elapsed for removal deductively: 0,100 seconds.
============================
ganglia> show
context:
Number of elements: 11
+-----------+----------+--------------+---------------------------------------------+...+
| type      | depth    | desiredDepth | derivations                                 |   |
+-----------+----------+--------------+---------------------------------------------+...+
| A         | 1        | 1            | [a: A]                                      |   |
| B         | infinity | infinity     | [([A->B]) ([A]): B]                         |   |
| X         | infinity | infinity     | [([(A->B)->X]) ([A->B]): X]                 |   |
| A->B      | infinity | infinity     | [fn v0 => [B]: A->B]                        |   |
| (A->B)->X | 1        | 1            | [fn v0 => [X]: (A->B)->X, cover: (A->B)->X] |   |
+-----------+----------+--------------+---------------------------------------------+...+

free variables: [a: A, cover: (A->B)->X]
scheduled constants to add: []
scheduled constants to remove: []

ganglia>
```

The infinite loop of naive implementations as mentioned in section 6.2.2.1 does not take place.

Chapter 9

Conclusion

9.1 Résumé

Many applications or parts of those can be treated as compositions and abstractions of services and information. Even if these are needed, so far only proprietary attempts are realised and published.

We considered the question whether and how such compositions can be done automatically. Our approach is guided by types as types offer an abstract and generic description of services and information. They are well understood and are for example used in programming languages and for theorem provers.

Answers to instances of this general question consume provably a huge amount of time. Too much for its applictions in practice. Moreover, it would also offer a vulnerable point just by its run time complexity.

To get the deduction logic tractable we focused on a fragment of this logic – a fragment weak enough to be efficiently computable, but yet strong enough to model properly our environment by simple types.

The proposed system comes close to the concept of traders. Its database contains the actual references to information and services. But additionally their deductive closure is carried along.

Both structures are affected by structural changes of the enviroment. They are indicated by changes in the database. We adapted and dualised the database for the closure in order to avoid redundant and expensive recomputations of the closure every time the content of the database changes. These operations are called `add` and `remove` (cf. the diagram on page 6).

The implemention ideas of both operations were detailed and proved to be optimal measured in the amount of accesses to persistent data structures.

The handling of questions for compositions of a given type is divided into two parts. The first constructs a selection for this type. We detailed the depth driven approach and considered another approach which ensures that the amount of information used for the composition is minimal. This could be suitable for environments with inaccurate information. The second part treats the representation for the questioner. We considered two implemented representation for lambda terms and compare them to each other and to representations of system F and as CoCoGraph. After all, the minimal lambda representation seems to be more useful.

9.2 Future work

We itemize some selected questions arosen during writing of this thesis.

Polymorphism. As indicated by references to functional and logical languages, parametric polymorphism might be interesting for example for nested contexts. Assume someone has a set of hotels and a function to enrich an individual hotel by its quality. To obtain a set of evaluated hotels a polymorphic function like SML's `map` might be purposeful.

To experiment with such an extension the current implementation offers a naive plugin for an additional term constructor `map` and an uniterated type constructor `list`, i.e. `list list A` is forbidden. Thereby the hotel problem can be tackled like:

```
 --   --         --         --
|    |  | |\  | |    |   | |  |
| -  |__| | \ | | -  |   | |__|
|__| |  | |  \| |__| |__ | |  |

Using polymorphic extension.
Enter a command, as example "help" for help.
ganglia> /* To be started with "java user/Ganglia -polymorphic" */
ganglia> /* or with "java -jar ganglia.jar -polymorphic" */
ganglia> add locator Position
[ OK ] "locator" was queued to adding.
ganglia> add enumerator Position>Hotel list
[ OK ] "enumerator" was queued to adding.
ganglia> add eval Hotel>EvalHotel
[ OK ] "eval" was queued to adding.
ganglia> dc
======== statistics ========
 Time elapsed ...
  ... to order terms and determinate borders:  0,060 seconds,
   thereby 3 terms were considered.
  ... to close context structurally:           0,030 seconds,
   thereby 9 items were considered.
  ... to close context ordinally:              0,000 seconds,
   thereby 3 items were considered.
----------------------------
 Total time elapsed for adding deductively:    0,090 seconds.
============================
ganglia> min EvalHotel list
Derivation is:
  map eval (enumerator locator): EvalHotel list

Its depth is 3 and
its size 7.
ganglia>
```

Quality of service. As affected in the introducing chapter 1.

Term of derivations. A weaker restriction to context stable derivations could expand the expressability. Such an extention could be achieved by *bounded derivations* Λ_k for some globally fixed parameter k. These are those terms $t \in \Lambda_k$ if and only if for all subterms s of t's normalform $|\,\mathrm{FV}(s)| \leq k$. Notice that Λ_0 equals the set of context stable normal forms.
Up to now every reasonable type corresponds to an individual contextitem. But within this extension up to approximately kn^k contextitems have to be assigned to an individual type whereby n measures the cardinality of the domain since contextitems needs to split for every assortment of at most k types out of n.

Set operations of contexts. In particular the union could be of interest. For example two entities might be interested in sharing information which can be modeled as a union of deductively closed contexts or parts of them.

Hierarchies on contexts. To simplify the management and for efficient operations contexts should be possible to arrange in hierarchies as introduced in section 1.2 by figure 1.4.

Bibliography

[Bara 01] Barabási, Albert-László: *Supplementary Material (for Parasitic Computing)*, 2001. Available at `http://www.nd.edu/~parasite/`.

[BFJB 01] Barabási, Albert-László, Vincent Freeh, Hawoong Jeaong and Jay Brockman: *Parasitic computing*. nature, 412, August 2001. Available at `http://www.nd.edu/~parasite/`.

[BrEi 04] Bry, François and Norbert Eisinger: *Lecture Notes: Übersetzerbau*. Technical Report, Ludwig-Maximilians-Universität München, PMS, 2004.

[Brui 72] Bruijn, Nicolaas G. De: *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. Indigationes Mathematicae, 34:381 – 392, 1972.

[Bry 04] Bry, François: *Lecture Notes: Informatik I*. Technical Report, Ludwig-Maximilians-Universität München, PMS, 2004.

[CGH 94] Cremers, Armin B., Ulrike Griefahn and Ralf Hinze: *Deduktive Datenbanken – eine Einführung aus der Sicht der logischen Programmierung*. Vieweg, 1994.

[ChKo 00] Chen, Guanling and David Kotz: *A Survey of Context-Aware Mobile Computing Research*. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000, http://www.cs.dartmouth.edu/reports/TR2000-381/ .

[DAS 01] Dey, Anind K., Gregory D. Abowed and Daniel Salber: *A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications*. context-aware computing in the Human-Computer Interaction (HCI), 16(2–4):97–166, 2001. `http://www.cs.berkeley.edu/~dey/context.html`.

[Deis 02] Deiser, Oliver: *Einführung in die Mengenlehre*. Springer-Verlag, 1st edition, 2002.

[dS 76] Swart, Harrie de: *Another intuitionistic completness proof*. The Journal of Symbolic Logic, 41:644 – 662, 1976. Reference Material for the lectures of A. Nerode given at the International Summer School on Logic, Algebra and Computation, Marktoberdorf, Germany, July 25 – August 6, 1989.

[GHJV 95] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Hind 97] Hindley, J. Roger: *Basic Simple Type Theory*. Cambridge University Press, 1997.

[Jung 95] Jungnickel, Dieter: *Codierungstheorie*. Spektrum Akademischer Verlag, 1995.

[KeEi 99] Kemper, Alfons and André Eickler: *Datenbanksysteme – Eine Einführung*. Oldenbourg-Verlag, 3rd, corrected edition, 1999.

[Krau 03] KRAUSE, MICHAEL: *Eine Sprache zur dynamischen Komposition von Kontextinformationen*. Master's thesis, Department of Computer Science at the University of Munich, August 2003.

[KSB 02] KÜPPER, AXEL, MICHAEL SCHIFFERS and THOMAS BUCHHOLZ: *A Framework for Context Provisioning in UMTS Networks*. Available from the authors, 2002.

[Lamp 90] LAMPING, JOHN: *An Algorithm for Optimal Lambda Calculus Reduction*. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16 – 30, January 1990.

[LP 03] LINNHOFF-POPIEN, CLAUDIA: *Lecture Notes: Verteilte Systeme*. Technical Report, Ludwig-Maximilians-Universität München, NM, 2003.

[Matt 00] MATTHES, RALPH: *Lambda Calculus: A Case Study for Inductive Definitions*. Available at `http://www.tcs.informatik.uni-muenchen.de/~matthes/works.html`, 2000.

[Matt 04] MATTHES, RALPH: *Non-strictly positive fixed-points for classical natural deduction*. Accepted for publication, available at `http://www.tcs.informatik.uni-muenchen.de/~matthes/works.html`, January 2004.

[Miln 78] MILNER, ROBIN: *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences, 17(3):348–375, 1978.

[Moor 65] MOORE, GORDON: *Cramming more components onto integrated circuits*. Electronics, 38(8), April 1965. `ftp://download.intel.com/research/silicon/moorespaper.pdf`.

[MyO' 84] MYCROFT, ALAN and RICHARD A. O'KEEFE: *A polymorphic type system for Prolog*. Artificial Intelligence, 23:295–307, 1984.

[OtWi 96] OTTMANN, THOMAS and PETER WIDMAYER: *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 3rd, revised edition, 1996.

[Papa 94] PAPADIMITRIOU, CHRISTOS: *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[Reis 90] REISCHUK, KARL RÜDIGER: *Einführung in die Komplexitätstheorie*. Teubner-Verlag, 1st edition, 1990.

[Stin 95] STINSON, DOUGLAS R.: *Cryptography - Theory and Practice*. CRC Press, Inc., 1995.

[Sun 04] SUN MICROSYSTEMS INC., `http://java.sun.com/j2se/1.5.0/docs/guide/collections/`: *The Collections Framework*, 2004.

[SuS 89] SUSE LINUX 9.0: *Maunal entry for `GNU make 3.8`*, 1989.

[TrSc 96] TROELSTRA, A. S. and H. SCHWICHTENBERG: *Basic Proof Theory*. Cambridge University Press, 1996.

[Weic 01] WEICH, KLAUS: *Improving Proof Search in Intuitionistic Propostional Logic*. PhD thesis, Department of Mathematics at the University of Munich, 2001.

[Weis 91] WEISER, MARK: *The Computer for the Twenty-First Century*. Scientific American, pages 94–110, September 1991. `http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html`.

Index