

INSTITUT FÜR INFORMATIK

DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN



Diplomarbeit

TERMINATION CHECKING FOR A DEPENDENTLY TYPED LANGUAGE

Dezember 2007

Autor: Karl Mehlretter
Aufgabensteller: Prof. Martin Hofmann
Betreuer: Dr. Andreas Abel

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Karl Mehlretter

Danksagung

Diese Diplomarbeit entstand am Lehrstuhl für Theoretische Informatik der Universität München. Vor allem danke ich meinem Betreuer Andreas Abel für seine tolle Unterstützung.

Desweiteren möchte ich mich bei meinen Eltern, bei Sandrine, bei Karin und Thomas und auch bei Andy, Robert und Christian für ihren Rückhalt während dieser nicht immer einfachen Zeit bedanken.

Inhalt

Abhängige Typen werden als Basis für viele interaktive Theorembeweiser eingesetzt und es gibt seit einiger Zeit Bemühungen, sie für allgemeine funktionale Programmiersprachen zu verwenden. Eine Sprache mit abhängigen Typen erlaubt es, in einem gemeinsamen Rahmen sowohl Programme zu schreiben als auch Beweise zu formalisieren.

Das aus der Programmierung bekannte *pattern matching* wurde als Mittel zur Definition von rekursiven Funktionen in diesen Systemen vorgeschlagen. Zwar ist *pattern matching* intuitiv und ausdrucksstark, doch muss sichergestellt werden, dass die so definierten Funktionen total sind.

Ein Aspekt einer totalen Funktion ist Terminierung: Jeder Auswertung der Funktion muss in endlicher Zeit möglich sein. Unendliche Objekte können unterstützt werden, indem jeweils nur Teile davon berechnet werden. Gültige Definitionen für unendliche Objekte müssen *produktiv* sein.

Da das Halteproblem unentscheidbar ist, kann das Ziel nur sein, möglichst viele Definitionen als terminierend zu erkennen. Es gibt verschiedene Ansätze zum Überprüfen der Terminierung. Oft können Funktionen, deren rekursive Aufrufe einem gewissen Schema folgen, vom System zugelassen werden. Das *size-change principle* umfasst einige dieser Schemata.

Ein weiterer Vorschlag ist der Einsatz von *sized types*. Information über die Größe von Argumenten wird im Typsystem verfolgt, um diese für das Erkennen der Terminierung zu nutzen.

Im Rahmen dieser Diplomarbeit wurde das System *Mugda* entwickelt und wird nachfolgend vorgestellt. Es basiert auf Typentheorie nach Martin-Löf und unterstützt induktive und coinduktive Typen. Rekursive Funktionen werden durch *pattern matching* definiert. Das Kriterium für Terminierung basiert auf dem *size-change principle*. Außerdem stellt die Sprache Elemente bereit, um dem Benutzer die Verwendung von *sized types* zu ermöglichen.

Abstract

Dependent types have been used at the core of many proof assistants, and more recently there have been efforts to extend their use to functional programming languages. Dependent type theories allow programming and reasoning in a common framework.

The *pattern matching* notation, known from traditional programming, has been proposed for defining recursive functions in such systems. While pattern matching is intuitive and powerful, it has to be ensured that the defined functions are total.

One aspect of totality is termination: the evaluation of a function at any argument must be computable in finite time. Infinite objects can be added to the language by computing only finite parts of them as necessary. Valid definitions of infinite objects need to be *productive*.

As the halteproblem is undecidable, the goal can only be to accept as many valid definitions as possible. There are many approaches to ensure termination. One is to allow the definition of functions where the recursive calls follow a certain scheme. The *size-change principle* subsumes some of these schemes.

Another approach is the use of *sized types*, where information about the height of arguments is tracked in the type system and is used to recognize definitions as terminating.

For this thesis, the system *Mugda* was developed and will subsequently be described. It is based on *Martin-Löf type theory* and supports inductive and coinductive types. Recursive functions are defined by pattern matching. Its termination criterion is based on the size-change principle. In addition, the language provides elements to enable the use of sized types.

Contents

1	Introduction	4
1.1	Overview	7
2	Mugda : Syntax and Semantics	8
2.1	Preliminaries	8
2.2	Syntax of Mugda	8
2.3	Semantics of Mugda expressions	10
2.3.1	Values	10
2.3.2	Signatures	11
2.3.3	Evaluation	12
2.4	Example programs	14
2.4.1	Identity function	14
2.4.2	Booleans	14
2.4.3	Natural numbers	15
2.4.4	Lists	15
2.4.5	Finitely branching trees	15
2.4.6	Vectors	15
2.4.7	Equality	16
2.4.8	Streams	17
3	Type-Checking	18
3.1	Scope-Checking	18
3.2	Bidirectional type-checking	18
3.3	Let declarations	22
3.4	Data type declarations	22
3.4.1	Checking data and constructor types	22
3.4.2	Strict positivity	23
3.4.3	Checking the whole declaration	26
3.5	Function declarations	26
3.5.1	Syntactic checks for patterns	27
3.5.2	Coverage of pattern matching	27
3.5.3	Preliminaries	27
3.5.4	Checking accessible part of patterns	28

3.5.5	Checking inaccessible patterns	29
3.5.6	Checking the whole declaration	29
3.6	Mugda programs	31
4	Termination-Checking	32
4.1	Motivation	32
4.2	Matrix notation	33
4.3	Relating pattern and expressions	33
4.4	Applying the size-change principle	37
4.5	Examples	39
4.5.1	Addition	39
4.5.2	Mutual even and odd	40
4.5.3	Brouwer ordinals	40
4.6	Extending the order	41
4.6.1	Examples	43
4.7	List reversion: Vectors to the rescue	44
5	Sized data types	46
5.1	Adding a Size type	46
5.1.1	Syntax and semantics	47
5.1.2	Type-Checking	48
5.1.3	Termination-Checking	48
5.2	Sized data type declarations	49
5.2.1	Examples	49
5.2.2	Checking sized data type declarations	50
5.3	Subtyping for size	51
5.4	Examples: sized inductive types	53
5.4.1	Natural number division	53
5.4.2	Sized Lists	55
5.4.3	Sized Brouwer ordinals	56
5.4.4	A higher-order function	56
5.5	Examples: Sized coinductive types	57
5.5.1	Sized Streams	57
5.5.2	Fibonacci stream	58
5.5.3	Equality of streams	58
5.5.4	Stream processors	59
5.6	Admissible recursive function declarations	60
5.6.1	Admissible type	62
5.6.2	Size pattern coverage	62
5.6.3	Admissibility criterion	63
5.7	Admissible corecursive declarations	64
5.7.1	Admissibility criterion	64
5.7.2	Fibonacci à la Haskell	65
5.8	On the necessity of subtyping	65

<i>CONTENTS</i>	3
5.9 Putting it all together	66
6 Conclusion	67
A Mugda implementation	69
A.1 Source file listing	69
A.2 Usage	70
A.3 Some implementation details	71

Chapter 1

Introduction

Dependent type theory. Dependent types are an important part of Martin-Löf type theory [ML84], which was conceived as a foundation for constructive mathematics.

Through the proofs-as-programs paradigm, they are fundamental to proof assistants like Coq [INR07], Lego [Pol94] and Twelf [PS99]. As programming languages receive more and more powerful type systems, there have been renewed efforts to use dependent types in programming languages: Cayenne [Aug98], Agda [Nor07] and Epigram [McB07] strive to be seen more as practical programming languages rather than logical frameworks.

Inductive families. Dependent type theory¹ is often described as an open theory: new constants can be added to the system. As a programming language, there is the need to define inductive types like lists and trees. For logical purposes, inductive predicates have to be defined.

Inductive families [Dyb94] offer a powerful general scheme for defining inductive types. A nice example for programming is matrix multiplication. One could define an inductive family of types `Mat` that is *indexed* by two natural numbers. The type for matrix multiplication could then be given as

```
mmult : Mat k m -> Mat m n -> Mat k n
```

which is not possible with a simple type system.

Pattern Matching. In traditional type theory, so-called elimination constants for each data type provide primitive recursion on objects. As an alternative, it was suggested in [Coq92] to allow definitions by pattern matching, a well-known concept from functional languages [Aug85].

¹for an introductory text, the reader is referred to [NPS90]

Pattern matching does allow for more readable and intuitive definitions, which is essential for type theory² to be useful as a programming languages. It has to be ensured that the pattern clauses cover all possible cases.

Termination Furthermore, pattern matching allows unrestricted recursion, and functions with non-terminating computations have to be disallowed for the following reasons:

- Proofs as programs need to be total functions to be correct.
- As types can depend on terms, computation is performed during type checking. Thus, for type checking to be decidable, non-terminating computation cannot be allowed.

To be more concrete, let us look at a simple recursive definition by pattern matching on a list in the functional programming Haskell:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

The only recursive call is `length xs`. The argument `xs` is a structural part of the input argument on the left hand side (`x:xs`). We conclude, if the list data type is well founded (i.e there are only finite lists), then the program `length` is guaranteed to terminate on all input lists.

Term-based termination. We note that we looked only at the defining clauses of the function to decide termination, its type declaration was irrelevant. Such methods are called *term-based* to differentiate them from the *type-based* methods discussed below.

When pattern matching for dependent types was introduced in [Coq92], the test that was outlined above was suggested as the criterion for termination: at least one argument needs to get *structurally smaller* in each recursive call. But not all defined functions have that clear *structurally recursive* form.

In [AA02], a decision procedure for a simply typed language was given that also handles lexicographic orderings on arguments and mutual definitions. The *size change principle* [LJBA01] subsumes this effort and also handles functions with so-called permuted arguments.

²Pattern matching is actually not a conservative extension to a theory with elimination constants. Pattern matching definitions can only be translated to a traditional type theory by adding the so-called *K Axiom* [HS95, GMM06].

Type-based termination. The above methods require that a declaration does not only need to be accepted by the *type-checker*, but also pass a separate syntactic *termination check*. In contrast, *type-based termination*³ refers to methods where the type-checker itself is in charge of checking termination: It is ensured by the typing rules.

The *sized types* approach is an instance of type-based termination: Inductive types are decorated with size annotations that denote an upper bound on the height of the objects inhabiting them. It is then checked that this size is getting smaller during recursive calls.

In comparison with term-based termination, sized types are more robust to syntactical changes to the program and the type system is able to use the information that certain functions are *size-preserving*. Some effort [BGP06] has been made to automatically infer these size annotations.

Infinite objects. Often there arises the need for potentially infinite objects, for example to model a continuous stream of network packages. Streams, infinite sequences of elements, are the prime example of *coinductive* types, where inhabitants can have *infinite height*.

While *corecursive definitions* that create or manipulate infinite objects are inherently non-terminating, they still should be *productive*. For productive definitions, looking at a finite portion of the object is well-defined. As for termination, there are syntactic tests to guarantee productivity [Coq93], but here again sized types offer a worthwhile alternative.

Mugda. Our work started with an investigation of how a sized type approach could be added to a system like Agda. The current version of Agda supports inductive families and mutual recursive function declarations by pattern matching. The Agda termination checker was, at that time, based on the work of [AA02].

In the description above, sized types sound like a very different approach compared to term-based termination. But, with inductive families and pattern matching, these approaches can be naturally combined.

As a result, the system Mugda was developed. It supports inductive families and pattern matching. As special features, the language supports coinductive definitions and also offers a built-in *Size type* to form size annotations. We outline here how termination and productivity checking of definitions is handled by Mugda :

- a syntactic termination checker based on the size-change principle is employed.
- in addition, the *Size type* can be used to create sized data types, enabling advantages of the sized type approach.

³for an introduction, see [Abe06]

- productivity of corecursive functions is also guaranteed by the Size type.
- the usage of the Size type is controlled by the type checker.

1.1 Overview

We introduce the syntax and semantics of the *Mugda* language in the next chapter, and show some example programs. The language described is missing the Size type that is later added in chapter 5.

Type-Checking for *Mugda* is presented in chapter 3. We cannot just focus on termination-checking because a termination-checker can easily be tricked by ill-typed declarations.

In the fourth chapter, a termination criterion for inductive function declarations based on the size-change principle is presented. Then we motivate and present an extension that increases its strength for definitions with nested patterns.

In chapter 5, the Size type is added to the language, which enables the formation of sized data types. It is shown how sized data types help to allow more recursive definitions pass the termination check of the third chapter. It also enables the declaration of a broad range of productive corecursive definitions to pass this check. Finally, we first motivate why the usage of size annotations needs to be restricted, and then give an *admissibility criterion* to achieve this.

In the conclusion, we take a look back and list ideas for future work. The appendix contains a description of the implementation of *Mugda*.

Chapter 2

Mugda : Syntax and Semantics

2.1 Preliminaries

First we define some common mathematical sets:

Natural numbers $\mathbb{N} = \{0, 1, \dots\}$ is the set of natural numbers.

Booleans $\mathbb{B} = \{\top, \perp\}$ is the set of Boolean values.

Power Set For some set B , $\mathcal{P}(B)$ is the power set of B .

Sequences For some set B , B^* is the set of finite sequences over B . The empty sequence is written as \diamond . Sequences are written as \vec{b} or $(b_1 \dots b_n)$. The length of a sequence is defined as $|(b_1 \dots b_n)| := n$.

2.2 Syntax of Mugda

Identifiers

We assume the following disjoint sets of identifiers:

$\mathbb{C} \ni c$	for constructors
$\mathbb{D} \ni D$	for data types
$\mathbb{F} \ni f, g$	for functions
$\mathbb{L} \ni l$	for global lets
$\mathbb{V} \ni x, y, i$	for variables

Elements of \mathbb{C} , \mathbb{D} , \mathbb{F} and \mathbb{L} will be called *constants*.

Expressions

EXPR $\ni e, A, B$::= $\lambda x. e$	abstraction
	$(x : A) \rightarrow B$	dependent function type
	$e e_1 \dots e_n$	application
	$\text{let } x : e_1 = A_1 \text{ in } e_2$	local let
	Set	universe of small types
	x	variable
	c	constructor name
	D	data type name
	f	function name
	l	let name

Expressions are the unified syntax of types and terms.

Patterns

PAT $\ni p$::= x	variable pattern
	$\mathbf{c} \vec{p}$	constructor pattern
	\underline{e}	inaccessible pattern

Telescopes

τ	::= \diamond	empty telescope
	$(x : A) \tau$	parameter
	$(+ x : A) \tau$	strictly positive parameter

Constructor definitions

$$\gamma ::= \mathbf{c} : A$$

Clause definitions

$$\text{CLAUSE } \ni \kappa ::= \mathbf{f} \vec{p} = e$$

For a clause $\mathbf{f} \vec{p} = e$, \vec{p} is called the right hand side and e the left hand side.

Recursive function definitions

$$\mu ::= \text{fun } \mathbf{f} : A \vec{\kappa}$$

Corecursive function definitions

$$\nu ::= \text{cofun } \mathbf{f} : A \vec{\kappa}$$

Declarations

DECL $\ni \delta$::=	data $D \tau : A \vec{\gamma}$	inductive data type
		codata $D \tau : A \vec{\gamma}$	coinductive data type
		let $l : A = e$	global let
		mutual $\vec{\mu}$	mutual fun
		mutual $\vec{\nu}$	mutual cofun

Mugda program

A Mugda program is a list of declarations $\vec{\delta}$.

Syntactic sugar

- $A \rightarrow B$ is short for $(x : A) \rightarrow B$ where x is some variable not occurring in B .
- In the examples, we write mutual fun $f : A \vec{\kappa}$ as $\text{fun } f : A \vec{\kappa}$.
- If it is clear from the context, \diamond is dropped. For example, \mathbf{c} is short for the pattern $\mathbf{c} \diamond$ when \mathbf{c} is a nullary constructor.

Prepend Telescope to expression

Ignoring the $+$ annotations, for a telescope $\tau = (x_1 : A_1) \dots (x_n : A_n)$ we write $\tau \rightarrow B$ for the expression $(x_1 : A_1) \rightarrow \dots \rightarrow (x_n : A_n) \rightarrow B$. For an empty telescope $\tau = \diamond$, $\tau \rightarrow B$ is meant to be the expression B . Accordingly, we write $\Theta \rightarrow B$ for $(x_1 : A_1) \rightarrow \dots \rightarrow (x_n : A_n) \rightarrow B$.

2.3 Semantics of Mugda expressions

Computation is already needed during type-checking for dependent types. Thus, we first need provide an evaluation that is used during type-checking.

2.3.1 Values

The semantics of Mugda contain the main ingredients of the type-checking algorithm of [Coq96]: Closures and generic values. Closures, already introduced in [Lan63], are used for explicit substitution. We simultaneously define values and environments:

Values

$\text{VAL} \ni v$	$::=$	$v \vec{v}$	application
		$\text{Lam } x e^\rho$	abstraction
		$\text{Pi } x v e^\rho$	dependent function space
		a	atomic value
$\text{AVAL} \ni a$	$::=$	k	generic value
		Set	universe of small types
		c	constructor name
		f	function name
		D	data name

A generic value $k \in \mathbb{N}$ represents the computed value of a variable during type-checking.

Environments

$\text{ENV} \ni \rho, \Gamma$	$::=$	\diamond	empty environment
		$\rho, x = v$	extended with binding

A closure e^ρ is a pair of an expression e and an environment ρ . The environment provides bindings for the free variables occurring in e . Values can be seen as partially evaluated expressions that may contain closures.

2.3.2 Signatures

A signature carries information about all declared constants. Each declaration adds newly defined constants to the signature, which is empty at the beginning.

Signature

A signature Σ is defined as a partial polymorphic mapping:

- $\Sigma : \mathbb{F} \rightarrow \text{VAL} \times \text{CLAUSE}^* \times \mathbb{B}$
mapping a function constant to its type (as a value), the clauses and a flag to indicate whether the clauses have been type-checked.
- $\Sigma : \mathbb{L} \rightarrow \text{EXPR} \times \text{VAL}$
mapping a global let constant to the expression and its type.
- $\Sigma : \mathbb{C} \rightarrow \text{VAL}$
mapping a constructor constant to its type.
- $\Sigma : \mathbb{D} \rightarrow \text{VAL} \times \mathbb{N}$
mapping a data type constant to its type and the number of parameters.

The empty signature Σ_0 is undefined on all arguments.

2.3.3 Evaluation

Now the evaluation of a closure e^ρ can be defined. A closed expression can be evaluated in an empty environment.

We simultaneously define evaluation as a function \searrow along with some helper functions. Most of these functions are partial, but they are total on well-typed expressions. A fixed signature Σ is assumed.

Evaluation

$$\begin{aligned}
\searrow &: \text{EXPR} \times \text{ENV} \rightarrow \text{VAL} \\
\searrow (\lambda x. e)^\rho &= \text{Lam } x \ e^\rho \\
\searrow ((x : A) \rightarrow B)^\rho &= \text{Pi } x \ v_A \ B^\rho \text{ where } v_A = \searrow A^\rho \\
\searrow (\text{let } x : A = e_1 \text{ in } e_2)^\rho &= \searrow e_2^{\rho, x=v_1} \text{ where } v_1 = \searrow e_1^\rho \\
\searrow (e \ e_1 \dots e_n)^\rho &= \text{app } v \ v_1 \dots v_n \text{ where } v = \searrow e^\rho, v_i = \searrow e_i^\rho \\
\searrow \text{Set}^\rho &= \text{Set} \\
\searrow c^\rho &= c \\
\searrow f^\rho &= f \\
\searrow l^\rho &= \searrow e^\diamond \text{ where } \Sigma \mid = (e, v_l) \\
\searrow x^\rho &= \text{lkup } \rho \ x
\end{aligned}$$

The closures in $\text{Lam } x \ e^\rho$ and $\text{Pi } x \ v_A \ B^\rho$ do not have a binding for x . The missing binding has to be provided before these closures can be evaluated. This might be a concrete value (for example during β -reduction) or a fresh generic value k .

Environment look-up

$$\begin{aligned}
\text{lkup} &: \text{ENV} \times \mathbb{V} \rightarrow \text{VAL} \\
\text{lkup } (\rho, x = v) \ x &= v \\
\text{lkup } (\rho, y = v) \ x &= \text{lkup } \rho \ x \text{ if } y \neq x
\end{aligned}$$

Application

does β -reduction and inductive function application:

$$\begin{aligned}
\text{app} &: \text{VAL} \times \text{VAL}^* \rightarrow \text{VAL} \\
\text{app } u \ \diamond &= u \\
\text{app } (u \ \vec{c}_1) \ \vec{c}_2 &= \text{app } u \ (\vec{c}_1, \vec{c}_2) \\
\text{app } (\text{Lam } x \ e^\rho) \ (v, \vec{v}) &= \text{app } v' \ \vec{v} \text{ where } v' = \searrow e^{\rho, x=v} \\
\text{app } f \ \vec{v} &= \text{app}_{\text{fun}} \ f \ \vec{v} \text{ if } f \text{ is a fun} \\
\text{app } v \ \vec{v} &= v \ \vec{v} \text{ otherwise}
\end{aligned}$$

In the following, as pattern matching can fail, the object \uparrow is used to indicate this.

Pattern matching

Pattern matching returns an environment that binds the variables in the patterns to values. When matching against a coinductive constructor, the value is *forced*:

$$\begin{aligned} \text{match} &: \text{ENV} \times \text{PAT} \times \text{VAL} \rightarrow \text{ENV} \cup \{\uparrow\} \\ \text{match } \rho (\mathbf{c} \vec{p}) v &= \text{match}_f \rho p v' \quad \text{if } \mathbf{c} \text{ is a coinductive constructor} \\ &\quad \text{and } v' = \text{force } v \\ \text{match } \rho p v &= \text{match}_f \rho p v \text{ otherwise} \end{aligned}$$

$$\begin{aligned} \text{match}_f &: \text{ENV} \times \text{PAT} \times \text{VAL} \rightarrow \text{ENV} \cup \{\uparrow\} \\ \text{match}_f \rho \underline{e} v &= \rho \\ \text{match}_f \rho x v &= \rho, x = v \\ \text{match}_f \rho (\mathbf{c} \diamond) \mathbf{c} &= \rho \\ \text{match}_f \rho (\mathbf{c} \vec{p}) (\mathbf{c} \vec{v}) &= \text{match}_{\text{list}} \rho \vec{p} \vec{v} \\ \text{match}_f \rho p v &= \uparrow \text{ otherwise} \end{aligned}$$

$$\begin{aligned} \text{match}_{\text{list}} &: \text{ENV} \times \text{PAT}^* \times \text{VAL}^* \rightarrow \text{ENV} \cup \{\uparrow\} \\ \text{match}_{\text{list}} \rho \diamond \diamond &= \rho \\ \text{match}_{\text{list}} \rho (p, \vec{p}) (v, \vec{v}) &= \text{match}_{\text{list}} \rho' \vec{p} \vec{v} \text{ where } \rho' = \text{match } \rho p v \neq \uparrow \\ \text{match}_{\text{list}} \rho p v &= \uparrow \text{ otherwise} \end{aligned}$$

Matching of a clause

For a single clause, if all patterns of a clause match against the argument values, then the right hand side can be evaluated:

$$\begin{aligned} \text{match}_{\text{cl}} &: \text{ENV} \times \text{PAT}^* \times \text{EXPR} \times \text{VAL}^* \rightarrow \text{VAL} \cup \{\uparrow\} \\ \text{match}_{\text{cl}} \rho \diamond e \vec{v} &= \text{app } v \vec{v} \text{ where } v = \searrow e^\rho \\ \text{match}_{\text{cl}} \rho (p, \vec{p}) e (v, \vec{v}) &= \text{match}_{\text{cl}} \rho' \vec{p} \vec{v} \text{ if } \rho' = \text{match } \rho p v \neq \uparrow \\ \text{match}_{\text{cl}} \rho \vec{p} e \vec{v} &= \uparrow \text{ otherwise} \end{aligned}$$

Matching of multiple clauses

Now we define how a sequence of clauses matches against the argument values. Each clause is tried until one is matched or there are no clauses left. It is assumed that clauses are distinct and complete, so that at most one will match:

$$\begin{aligned} \text{match}_{\text{cls}} &: \text{CLAUSE}^* \times \text{VAL}^* \rightarrow \text{VAL} \cup \{\uparrow\} \\ \text{match}_{\text{cls}} \diamond \vec{v} &= \uparrow \\ \text{match}_{\text{cls}} ((f \vec{p} = e) \vec{\gamma}) \vec{v} &= v \text{ if } v = \text{match}_{\text{cl}} \diamond \vec{p} e \vec{v} \neq \uparrow \\ \text{match}_{\text{cls}} ((f \vec{p} = e) \vec{\gamma}) \vec{v} &= \text{match}_{\text{cls}} \vec{\gamma} \vec{v} \text{ otherwise} \end{aligned}$$

Note that matching can fail, even when the coverage of all clauses is complete: there can be too few arguments. Furthermore, generic values do not match against a constructor pattern.

Now the reduction behaviour of recursive and corecursive functions can be described. Only functions whose clauses have been type-checked (and later termination-checked) will trigger reduction.

Inductive function application

We always reduce application of an inductive function if possible:

$$\begin{aligned} \text{app}_{\text{fun}} &: \mathbb{V} \times \text{VAL}^* \rightarrow \text{VAL} \\ \text{app}_{\text{fun}} f \vec{v} &= v \text{ if } \Sigma f = (t, \vec{\gamma}, \top) \text{ and } \text{match}_{\text{cls}} \vec{\gamma} \vec{v} = v \neq \uparrow \\ \text{app}_{\text{fun}} f \vec{v} &= f \vec{v} \text{ otherwise} \end{aligned}$$

Corecursive unrolling

As said above, an application of a corecursive definition is lazily unrolled when needed. `force` tries to unroll a corecursive application until a constructor appears:

$$\begin{aligned} \text{force} &: \text{VAL} \rightarrow \text{VAL} \\ \text{force } f &= \text{force } (f \diamond) \\ \text{force } (f \vec{v}) &= \text{force } v \text{ where } v = \text{match}_{\text{cls}} \vec{\gamma} \vec{v} \neq \uparrow \\ &\quad \text{if } \Sigma f = (t, \vec{\gamma}, \top) \text{ and } f \text{ is a cofun} \\ \text{force } v &= v \text{ otherwise} \end{aligned}$$

2.4 Example programs

2.4.1 Identity function

Without defining data types, we still can define some non-recursive functions. One example is the identity mapping. `Mugda` is monomorphic, so the type has to be supplied as the first argument:

```
let id : (A : Set) → A → A = λ A. λ a. a
```

2.4.2 Booleans

The type of Booleans is introduced with:

```
data Bool : Set
  tt : Bool
  ff : Bool
```

An if-then-else construct can be defined by pattern matching:

```

fun ite : (A : Set) → Bool → A → A → A
  ite A tt a b = a
  ite A ff a b = b

```

2.4.3 Natural numbers

The type of natural numbers:

```

data Nat : Set
  zero : Nat
  succ : Nat → Nat

```

The addition function can be defined recursively by pattern matching:

```

fun add : Nat → Nat → Nat
  add x zero = x
  add x (succ y) = succ (add x y)

```

2.4.4 Lists

Lists are an example of a parameterized data type:

```

data List (+ A : Set) : Set
  nil : List A
  cons : A → List A → List A

```

2.4.5 Finitely branching trees

The following declaration introduces finitely branching trees where nodes and leafs carry elements of type A .

```

data Tree (+ A : Set) : Set
  node : A → List (Tree A) → Tree A

```

A leaf is a node with an empty list of successors. In `node`, the recursive argument `Tree A` appears as a parameter to `List`. This will be allowed because the parameter of `List` is *strictly positive*.

2.4.6 Vectors

Now things get more interesting. Vectors are an example of an inductive family of types.

```

data Vec (+ A : Set) : Nat → Set
  nil : Vec A zero
  cons : (n : Nat) → A → Vec A n → Vec A (succ n)

```

Next let us define the `head` function that returns the first element of a vector. This operation should only be allowed for a non-empty vector. With dependent types, we can express this in the type signature:

```
fun head : (A : Set) → (n : Nat) → Vec A (succ n) → A
```

Advancing to the clause definition, the usage of inaccessible patterns will become clear. First, pattern matching against the vector argument is needed:

```
head ?? (cons B m x xl) = x
```

It is of note that a clause for the `nil` case is not needed, because `nil` does not belong to a type of the form `Vec A (succ n)`. Now, what about those question marks above? One might be inclined to use *non-linear* pattern variables:

```
head B m (cons B m x xl) = x
```

But really, the system need not check at run-time that the values at the corresponding arguments are equal – it is guaranteed for a well-typed program.

So to capture the notion that the first two arguments are automatically instantiated by matching against a constructor, the inaccessible pattern notation [Nor07, GMM06] is used. The final definition is:

```
fun head : (A : Set) → (n : Nat) → Vec A (succ n) → A
  head B m (cons B m x xl) = x
```

2.4.7 Equality

The following *predicate* is called Martin-Löf equality, another important example of a type that can be defined as an inductive family:

```
data Eq (A : Set) : A → A → Set
  refl : (a : A) → Eq A a a
```

where the single constructor `refl` states reflexivity of equality.

As a simple example, the program

```
let proof : (x : Nat) → Eq Nat (add x zero) x
  = λ y. refl Nat y
```

can now be seen as a proof of the mathematical proposition $\forall x \in \mathbb{N} : x + 0 = x$. Without having formally introduced type-checking, we want to provide some intuition on issues that arise while checking such declarations.

The above example passes the type-checker because `add x zero` reduces to `zero`, and which is – for the type system – equal to itself. Now, to prove $\forall x \in \mathbb{N} : 0 + x = x$, more effort is needed. The following will not be accepted:

```
let proof2 : (x : Nat) → Eq Nat (add zero x) x
    λ y. refl Nat y
```

The type-checker is only able to “see” definitional equality. As addition was defined by recursion on the second argument, `add zero x` does not reduce to `x` during type-checking.

More technically, the generic value introduced for `y` – that “moves up” to the type – neither matches against the constructor pattern `zero` in the first clause nor against the constructor pattern `succ y` in the second clause of `add`.

So what is needed is a recursive proof, by manually doing case distinction:

```
fun eqSucc : (x : Nat) → (y : Nat) → Eq Nat x y → Eq Nat (succ x) (succ y)
    eqSucc x x (refl Nat x) = refl Nat (succ x)
fun proof2 : (x : Nat) → Eq Nat (add zero x) x
    proof2 zero = refl Nat zero
    proof2 (succ x) = eqSucc (add zero x) x (proof2 x)
```

Recursive proofs need to be total functions to be logically valid.

2.4.8 Streams

Next we introduce streams as an example of a coinductive type. Coinductive types are not required to be well-founded, i.e. their inhabitants do not need to have finite height.

```
codata Stream : Set
    cons : Nat → Stream → Stream
```

For less clutter in later examples, we focus here on streams of natural numbers, although they could be parameterized just like lists. As `Stream` has only one constructor, there are no `Stream` objects of finite height.

A stream of zeroes can be declared by the following corecursive declaration:

```
cofun zeroes : Stream
    zeroes = cons zero zeroes
```

The first element of a stream can be computed by the function `head`:

```
fun head : Stream → Nat
    head (cons x xs) = x
```

As expected $\searrow (\text{head zeroes})^\circ = \text{zero}$ because `head` triggers an unfolding. This evaluation is well-defined because `zeroes` is productive.

Chapter 3

Type-Checking

The starting point for this chapter was the simple type-checking algorithm for dependent types that is given in [Coq96].

In the following, we will introduce judgments for type-checking. A rule of a judgment has the general form

$$\frac{A_1 \quad \dots \quad A_n}{B} c$$

where the A_i are the premises and B is the conclusion. All these judgments given in this work are *algorithmic*: a deterministic algorithm that builds a derivation *bottom-up* is directly induced by the rules. If more than one rule is applicable, the first one (in reading direction – left to right, top to bottom) must be chosen.

3.1 Scope-Checking

In the semantics of the previous section, looking up in an environment or in the signature was expected never to fail. This is only guaranteed for well-scoped programs, where identifiers are always *in scope*.

We deal with this problem here by postulating that type-checking fails if lookup in an environment or signature is not defined. In an actual implementation (see appendix), the use of identifiers can already be checked after parsing.

3.2 Bidirectional type-checking

A standard technique for dependent types called *bidirectional type-checking* is used. Intuitively, this means that the type-checker has two modes: one for checking that an expression has a certain type and one for checking that an expression is correct while inferring its type.

The following three judgments are defined simultaneously:

check type checks that A denotes a valid type (figure 3.3):

$$k; \rho; \Gamma \vdash A \text{ Type} \subseteq \mathbb{N} \times \text{ENV} \times \text{ENV} \times \text{EXPR}$$

check expression checks that e has the type v (figure 3.1):

$$k; \rho; \Gamma \vdash e \Leftarrow v \subseteq \mathbb{N} \times \text{ENV} \times \text{ENV} \times \text{EXPR} \times \text{VAL}$$

infer type checks that e is correct and infers its type value v (figure 3.2):

$$k; \rho; \Gamma \vdash e \Rightarrow v : \mathbb{N} \times \text{ENV} \times \text{ENV} \times \text{EXPR} \rightarrow \text{VAL}$$

The environment ρ will be used to bind fresh generic values to variables, and Γ will bind the type value corresponding to these generic values.

As noted, in checking mode, the type-checker might have to infer the type of the expression and then verify that the inferred type value is equal to the one that is being checked against: *Equality* (also called *convertibility*) between two values is needed:

Equality of values A simple equality derived from [Coq96] is presented. Regarding infinite objects, a corecursive definition should be equal to its unfolding. We try to approximate this by allowing the type-checker to unroll (force) a corecursive definition. Unlimited unfolding cannot be allowed, thus we keep track of unfolding and only allow one side of two to be unrolled.

The *force history* set $\mathbb{F} = \{\mathbf{L}, \mathbf{R}, \mathbf{N}\}$ is introduced to keep track if the left or right side (v_2) has been forced, or no side yet. We will return to the subject of equality for infinite objects in section 5.5.3.

Two judgments (figure 3.4) are defined simultaneously, the latter operating on already unrolled values:

$$f; k \vdash v_1 \Leftrightarrow v_2 \subseteq \mathbb{F} \times \mathbb{N} \times \text{VAL} \times \text{VAL}$$

$$f; k \vdash v_1 \leftrightarrow v_2 \subseteq \mathbb{F} \times \mathbb{N} \times \text{VAL} \times \text{VAL}$$

After having defined the basic building blocks, type-checking individual declarations is next. For readability, we formalize these as imperative algorithms rather than judgments. When a declaration passes the type-check, the signature can be expanded.

Type-correctness of empty signature:

The empty signature Σ_0 is *type-correct*.

$$\begin{array}{c}
\text{CHK-LET} \frac{k; \rho; \Gamma \vdash A \mathbf{Type} \quad k; \rho; \Gamma \vdash e_1 \Leftarrow \searrow A^p \quad k; \rho, x = \searrow e_1^p; \Gamma, x = \searrow A^p \vdash e_2 \Leftarrow v}{k; \rho; \Gamma \vdash \text{let } x : A = e_1 \text{ in } e_2 \Leftarrow v} \\
\text{CHK-LAM} \frac{k + 1; \rho, x = k; \Gamma, x = v_A \vdash e \Leftarrow \searrow t^{\rho, y=k}}{k; \rho; \Gamma \vdash \lambda x. e \Leftarrow \text{Pi } y v_A t^p} \\
\text{CHK-PI} \frac{k; \rho; \Gamma \vdash A \Leftarrow \mathbf{Set} \quad k + 1; \rho, x = k; \Gamma, x = \searrow A^p \vdash B \Leftarrow \mathbf{Set}}{k; \rho; \Gamma \vdash (x : A) \rightarrow B \Leftarrow \mathbf{Set}} \\
\text{CHK-INF} \frac{k; \rho; \Gamma \vdash e \Rightarrow v_2 \quad \mathbf{N}; k \vdash v_2 \Leftrightarrow v_1}{k; \rho; \Gamma \vdash e \Leftarrow v_1}
\end{array}$$

Figure 3.1: Checking expressions

$$\begin{array}{c}
\text{INF-APP-I} \frac{k; \rho; \Gamma \vdash e_1 \Rightarrow \text{Pi } x v_A B^p \quad k; \rho; \Gamma \vdash e_2 \Leftarrow v_A}{k; \rho; \Gamma \vdash e_1 e_2 \Rightarrow \searrow B^{\rho, x = \searrow e_2^p}} \\
\text{INF-APP-II} \frac{k; \rho; \Gamma \vdash (e_1 e_2) \vec{e} \Rightarrow v}{k; \rho; \Gamma \vdash e_1 (e_2 \vec{e}) \Rightarrow v} \quad \text{INF-VAR} \frac{v = \text{lkup } \Gamma x}{k; \rho; \Gamma \vdash x \Rightarrow v} \\
\text{INF-FUN} \frac{\Sigma f = (v, \vec{\gamma}, b)}{k; \rho; \Gamma \vdash f \Rightarrow v} \quad \text{INF-CON} \frac{\Sigma c = v}{k; \rho; \Gamma \vdash c \Rightarrow v} \\
\text{INF-DATA} \frac{\Sigma D = (v, n)}{k; \rho; \Gamma \vdash D \Rightarrow v} \quad \text{INF-DEF} \frac{\Sigma l = (e, v)}{k; \rho; \Gamma \vdash l \Rightarrow v}
\end{array}$$

Figure 3.2: Inferring type of expressions

$$\begin{array}{c}
\frac{k; \rho; \Gamma \vdash A \mathbf{Type} \quad k + 1; \rho, x = k; \Gamma, x = \searrow A^p \vdash B \mathbf{Type}}{k; \rho; \Gamma \vdash (x : A) \rightarrow B \mathbf{Type}} \\
\frac{}{k; \rho; \Gamma \vdash \mathbf{Set Type}} \quad \frac{k; \rho; \Gamma \vdash t \Leftarrow \mathbf{Set}}{k; \rho; \Gamma \vdash t \mathbf{Type}}
\end{array}$$

Figure 3.3: Type judgment

$$\begin{array}{c}
\text{EQ-FORCE} \frac{\begin{array}{l} \mathbf{L}; k \vdash \text{force } v_1 \leftrightarrow v_2 \quad \text{if force } v_1 \neq v_1, \text{force } v_2 = v_2, f \neq \mathbf{R} \\ \mathbf{R}; k \vdash v_1 \leftrightarrow \text{force } v_2 \quad \text{if force } v_1 = v_1, \text{force } v_2 \neq v_2, f \neq \mathbf{L} \\ f; k \vdash v_1 \leftrightarrow v_2 \quad \text{otherwise} \end{array}}{f; k \vdash v_1 \leftrightarrow v_2} \\
\\
\text{EQ-APP} \frac{f; k \vdash v \leftrightarrow w \quad f; k \vdash v_j \leftrightarrow w_j \text{ for all } j \in \{1 \dots n\}}{f; k \vdash v v_1 \dots v_n \leftrightarrow w w_1 \dots w_n} \\
\\
\text{EQ-PI} \frac{f; k \vdash v_1 \leftrightarrow v_2 \quad f; k+1 \vdash \searrow b_1^{\rho, x_1=k} \leftrightarrow \searrow b_2^{\Gamma, x_2=k}}{f; k \vdash \text{Pi } x_1 v_1 b_1^{\rho} \leftrightarrow \text{Pi } x_2 v_2 b_2^{\Gamma}} \\
\\
\text{EQ-LAM} \frac{f; k+1 \vdash \searrow e_1^{\rho, x_1=k} \leftrightarrow \searrow e_2^{\Gamma, x_2=k}}{f; k \vdash \text{Lam } x_1 e_1^{\rho} \leftrightarrow \text{Lam } x_2 e_2^{\Gamma}} \quad \text{EQ-ATOM} \frac{}{f; k \vdash a \leftrightarrow a}
\end{array}$$

Figure 3.4: equality checking

3.3 Let declarations

Checking a global let declaration is pretty straightforward:

Algorithm

For $\delta = \text{let } l : A = e$ and a type-correct signature Σ , if

1. $1; \diamond; \diamond \vdash A \text{ Type}$
2. $v_A := \searrow A^\diamond$
3. $1; \diamond; \diamond \vdash e \Leftarrow v_A$
4. $\Sigma' := \Sigma \cup \{l \mapsto (e, v_A)\}$

then Σ' is a type-correct signature.

3.4 Data type declarations

We have to check that data type declarations follow the scheme of inductive families [Dyb94]. Actually, we here stay closer to the current version of Agda, namely the strict positivity test is more lenient. In addition, Mugda supports coinductive types [Coq93, Gim98] with the `codata` construct.

3.4.1 Checking data and constructor types

First, a valid data type declarations needs to have the syntactic form outlined in figure 3.5, so that the parameters $p_i \dots p_n$ in the result type of every constructor exactly match those of the telescope.

$$\begin{aligned}
 &\text{data } D (p_1 : P_1) \dots (p_n : P_n) : \Theta \rightarrow \text{Set} \\
 &\quad \mathbf{c}_1 : \Delta_1 \rightarrow D p_1 \dots p_n t_1^1 \dots t_m^1 \\
 &\quad \dots \\
 &\quad \mathbf{c}_k : \Delta_k \rightarrow D p_1 \dots p_n t_1^k \dots t_m^k
 \end{aligned}$$

Figure 3.5: Syntactic valid form of data declaration

Figure 3.7 shows the check for the data type, and figure 3.8 shows the check for each constructor type. As the types will depend on the parameters, we will have to prepend the telescope when checking them. For example, in the judgment $k; \rho; \Gamma; n \vdash A \text{ DataType}$ we have

- As in previous judgments, k is the next fresh generic value, and ρ and Γ are environments for the free variables.
- n is a fixed parameter that will be set to the length of the telescope $\tau = (p_1 : P_1) \dots (p_n : P_n)$.

- A is the expression that is left to be checked.

So for the declaration in figure 3.5, it is checked that

$$1; \diamond; \diamond; |\tau| \vdash \tau \rightarrow \Theta \rightarrow \mathbf{Set\ DataType}$$

holds.

The parameter types P_i can be arbitrary types, while the types in indices Θ and the constructor arguments in Δ_i need to be small types, i.e. of type \mathbf{Set} . Otherwise the data type \mathbf{V} in figure 3.6, adapted from [Coq92], would be accepted. Then `loop` would pass the termination-check of the next chapter. In a language with a *predicative* hierarchy of universes like Agda [Nor07], the type \mathbf{V} is not accepted as a small type, but as an inhabitant of a higher level (\mathbf{Set}_1). Then `loop` would not be type-correct.

```

data V : Set
  v : ((A : Set) → (A → A)) → V
fun loop : V → Set
  loop (v f) = loop (f V (v f))
let diverge : Set = loop (v id)

```

Figure 3.6: Invalid data type \mathbf{V}

$$\frac{
\begin{array}{l}
\text{if } k < n \text{ then } k; \rho; \Gamma \vdash A \mathbf{Type} \\
\text{if } k \geq n \text{ then } k; \rho; \Gamma \vdash A \Leftarrow \mathbf{Set} \\
k + 1; \rho, x = k; \Gamma, x = \searrow A^p; n \vdash B \mathbf{DataType}
\end{array}
}{
k; \rho; \Gamma; n \vdash (x : A) \rightarrow B \mathbf{DataType}
}$$

$$\frac{}{
k; \rho; \Gamma; n \vdash \mathbf{Set\ DataType}
}$$

Figure 3.7: Data-type judgment

3.4.2 Strict positivity

In a telescope, each $(p_i : P_i)$ can be written as $(+ p_i : P_i)$ to denote strict positivity.

Positive parameters

For D , we define the set of positive parameter indices $\text{pos}(D)$:

$$\text{pos}(D) \subseteq \mathbb{N} := \{i \mid p_i \text{ is declared strictly positive}\}$$

$$\frac{\text{if } k \geq n \text{ then } k; \rho; \Gamma \vdash A \Leftarrow \text{Set} \quad k+1; \rho, x = k; \Gamma, x = \searrow A^\rho; n \vdash B \text{ ConType}}{k; \rho; \Gamma; n \vdash (x : A) \rightarrow B \text{ ConType}}$$

$$\frac{k; \rho; \Gamma \vdash v \Leftarrow \text{Set}}{k; \rho; \Gamma; n \vdash v \text{ ConType}}$$

Figure 3.8: Constructor type judgment

We require that data type declarations are strictly positive. Otherwise there are inconsistencies [PM93], i.e. non-terminating terms could be constructed that would fail the check of the next chapter. Roughly, its meaning is: In every constructor argument, the data type to be defined is not allowed to occur in a function domain or in an application.

An exception to this that is allowed in *Mugda* is that parameters that are for themselves strictly positive do preserve strict positivity. This allows the definition of the *Tree* type in section 2.4.5 to pass the strict positivity test. In that regard, *Mugda*'s data types are closer to the *strictly positive families* described in [MAG07]. For the strict positivity test of a constructor (figure 3.11) the non-occurrence (figure 3.9) and the strict positive occurrence (figure 3.10) of an atomic value is needed.

$$\frac{k \vdash a \text{ nocc } v_A \quad k+1 \vdash a \text{ nocc } \searrow B^{\rho, x=k}}{k \vdash a \text{ nocc } \text{Pi } x v_A B^\rho} \quad \frac{k+1 \vdash a \text{ nocc } \searrow e^{\rho, x=k}}{k \vdash a \text{ nocc } \text{Lam } x e^\rho}$$

$$\frac{k \vdash a \text{ nocc } v \quad k \vdash a \text{ nocc } v_j \text{ for all } j \in \{1 \dots n\}}{k \vdash a \text{ nocc } v v_1 \dots v_n} \quad \frac{a \neq a'}{k \vdash a \text{ nocc } a'}$$

Figure 3.9: Non-occurrence of atomic value a

Example : Untyped lambda calculus

Figures 3.12 and 3.13 show two encodings of untyped lambda calculus that are rejected. The data type *Term* is rejected in figure 3.12, and in figure 3.13, *Fun* is rejected because its parameter *A* is declared as strictly positive, but is not. In both example, *app* is non-recursive and would pass the termination check of the next chapter.

$$\begin{array}{c}
\frac{k \vdash a \text{ nocc } v_A \quad k+1 \vdash a \text{ spos } \searrow B^{\rho, x=k}}{k \vdash a \text{ spos } \text{Pi } x v_A B^{\rho}} \quad \frac{k+1 \vdash a \text{ spos } \searrow e^{\rho, x=k}}{k \vdash a \text{ spos } \text{Lam } x e^{\rho}} \\
\\
\frac{\begin{array}{l} k \vdash a \text{ nocc } v_j \text{ for all } j \in \{1 \dots m\}, j \notin \text{pos}(D) \\ k \vdash a \text{ spos } v_j \text{ for all } j \in \text{pos}(D) \end{array}}{k \vdash a \text{ spos } D v_1 \dots v_m} \\
\\
\frac{k \vdash a \text{ spos } v \quad k \vdash a \text{ nocc } v_j \text{ for all } j \in \{1 \dots n\}}{k \vdash a \text{ spos } v v_1 \dots v_n} \\
\\
\frac{}{k \vdash a \text{ spos } a} \quad \frac{k \vdash a \text{ nocc } v}{k \vdash a \text{ spos } v}
\end{array}$$

Figure 3.10: Strictly positive occurrence of atomic value a

$$\frac{k \vdash a \text{ spos } v_A \quad k+1 \vdash a \text{ sposc } \searrow B^{\rho, x=k}}{k \vdash a \text{ sposc } \text{Pi } x v_A B^{\rho}} \quad \frac{}{k \vdash a \text{ sposc } D \vec{v}}$$

Figure 3.11: strict positivity test for constructor

```

data Term : Set
  abs : (Term → Term) → Term
fun app : Term → Term → Term
  app (abs f) y = f y
let omega : Term = abs (λ x. app x x)
let diverge : Term = app omega omega

```

Figure 3.12: Encoding of untyped lambda calculus

```

data Fun (+A : Set) : Set
  fn : (A → A) → Fun A
data Term : Set
  abs : Fun Term → Term
fun app : Term → Term → Term
  app (abs (fn Term f)) y = f y
let omega : Term = abs (fn Term (λ x. app x x))
let diverge : Term = app omega omega

```

Figure 3.13: Another encoding of untyped lambda calculus

3.4.3 Checking the whole declaration

Algorithm

For declaration $\delta = \text{data } D \tau : A \vec{\gamma}$ and type-correct signature Σ ,
if

1. δ follows form of figure 3.5
2. $n := |\tau|$
3. $1; \diamond; \diamond; n \vdash \tau \rightarrow A \text{ \textbf{DataType}}$
4. $v_D := \searrow (\tau \rightarrow A)^\diamond$
5. $\Sigma' := \Sigma \cup \{D \mapsto (v_D, n)\}$
6. Using Σ' , for every constructor declaration $c_i : B_i \in \vec{\gamma}$:
 - (a) $1; \diamond; \diamond; n \vdash \tau \rightarrow B_i \text{ \textbf{ConType}}$
 - (b) $v_i := \searrow (\tau \rightarrow B_i)^\diamond$
 - (c) $1 \vdash k \text{ \textbf{sposc}} v_i$ for every $j \in \text{pos}(D)$
 - (d) $1 \vdash D \text{ \textbf{sposc}} v_i$
7. $\Sigma'' := \Sigma' \cup \bigcup_i \{c_i \mapsto v_i\}$

then Σ'' is a type-correct signature.

A codata declaration is checked just the same.

3.5 Function declarations

Overview

We need to check each clause $f \vec{p} e$ of a function separately against the declared type. So for each clause, we have to check the patterns \vec{p} and then the right hand side e . Checking the patterns \vec{p} will yield an environment for the free variables in e .

Because of the inaccessible patterns, we follow [Nor07] and check \vec{p} in two phases. In the first phase, we will skip inaccessible patterns, and only check the *accessible part* of the patterns. The inaccessible patterns will be represented by fresh *flexible* generic values. These flexible generic values will be *instantiated* to concrete values when checking constructor patterns. Then in the second phase, it is verified that the expressions of the inaccessible patterns are equal to those instantiated in the first phase, and finally the right hand side e can be checked.

3.5.1 Syntactic checks for patterns

We first outline some syntactic requirements on the patterns:

- all patterns of a clause are linear - all variables occur only once in the accessible parts.
- variable bindings in the right hand side do not *shadow* the variable patterns.
- all clauses should have the same number of patterns.

This can already be checked during scope-checking. The last two items are important for the syntactic termination-checker described in the following chapter.

3.5.2 Coverage of pattern matching

Also not covered here is very important for logical consistency: It is necessary to check that the clauses cover all possible cases. This is quite complicated [Nor07, SP03] for inductive families and will not be described here. In addition, due to our “try all clauses” semantics, clauses should not overlap.

3.5.3 Preliminaries

Converting pattern to value

During type-checking of the accessible patterns, patterns will be converted to values because the type may depend on them:

$$\begin{aligned}
\Downarrow _ : \mathbb{N} \times \text{PAT} &\rightarrow \text{VAL} \\
\Downarrow^k p &= v \text{ where } \text{p2v } k p = (v, k') \\
\text{p2v} : \mathbb{N} \times \text{PAT} &\rightarrow (\text{VAL}, \mathbb{N}) \\
\text{p2v } k x &= (k, k + 1) \\
\text{p2v } k (\mathbf{c} \diamond) &= (\mathbf{c}, k) \\
\text{p2v } k (\mathbf{c} \vec{p}) &= (\mathbf{c} \vec{v}, k') \text{ where } (\vec{v}, k') = \text{p}_s\text{2v}_s k \vec{p} \\
\text{p2v } k \underline{e} &= (k, k + 1) \\
\text{p}_s\text{2v}_s : \mathbb{N} \times \text{PAT}^* &\rightarrow (\text{VAL}^*, \mathbb{N}) \\
\text{p}_s\text{2v}_s k \diamond &= (\diamond, k) \\
\text{p}_s\text{2v}_s k (p, \vec{p}) &= (k'', (v, \vec{v})) \text{ where} \\
&\quad (v, k') = \text{p2v } k p \\
&\quad (\vec{v}, k'') = \text{p}_s\text{2v}_s k' \vec{p}
\end{aligned}$$

Both pattern variables and inaccessible patterns are converted to ascending fresh generic values, just as later during checking of the pattern.

Instantiating by successful *unification* (figure 3.14) will yield a substitution for flexible generic values:

Substitutions

A substitution is a list $\sigma : (\mathbb{N} \times \text{VAL})^*$ and denotes a partial mapping of generic values to values. \diamond is the empty substitution. $\sigma[v/k]$ is the substitution where k is mapped to v and others according to σ .

Application of a substitution

Now we define simultaneously the application of a substitution on values and environments:

$$\begin{aligned}
\{\} &: (\mathbb{N} \times \text{VAL})^* \times \text{VAL} \rightarrow \text{VAL} \\
\sigma\{k\} &= v \text{ if } (k, v) \in \sigma \\
\sigma\{v \ v_1 \dots v_n\} &= \sigma\{v\} \ \sigma\{v_1\} \dots \sigma\{v_n\} \\
\sigma\{\text{Pi } x \ v_A \ B^\rho\} &= \text{Pi } x \ \sigma\{v_A\} \ B^{\sigma\{\rho\}} \\
\sigma\{\text{Lam } x \ e^\rho\} &= \text{Lam } x \ e^{\sigma\{\rho\}} \\
\sigma\{v\} &= v \text{ otherwise} \\
\{\} &: (\mathbb{N} \times \text{VAL})^* \times \text{ENV} \rightarrow \text{VAL} \\
\sigma\{\diamond\} &= \diamond \\
\sigma\{(x, v)\rho\} &= (x, \sigma\{v\})\sigma\{\rho\}
\end{aligned}$$

Composition of substitutions

Next is the composition $\text{comp } \sigma_1 \sigma_2$ of two substitutions σ_1 and σ_2 , with the intended meaning that the equation $\text{comp } \sigma_1 \sigma_2\{v\} = \sigma_2\{\sigma_1\{v\}\}$ holds:

$$\begin{aligned}
\text{comp} &: (\mathbb{N} \times \text{VAL})^* \times (\mathbb{N} \times \text{VAL})^* \rightarrow (\mathbb{N} \times \text{VAL})^* \\
\text{comp } ((k_1, v_1) \dots (k_n, v_n)) \sigma_2 &= ((k_1, \sigma_2\{v_1\}) \dots (k_n, \sigma_2\{v_n\})) \sigma_2
\end{aligned}$$

It is assumed that the domains of the substitutions are disjoint and that there are no occurrences of the generic values of σ_1 in the values of the codomain of σ_2 .

3.5.4 Checking accessible part of patterns

The main work of the first phase is the judgment

$$(k, \xi, \sigma, \rho, \Gamma) \vdash p \text{ chkP } v \Rightarrow (k', \xi', \sigma', \rho', \Gamma'), v'$$

where

- k denotes the next fresh generic value
- ρ and Γ are environments, again providing for a variable a value and its type.
- $\xi \subseteq \mathbb{N} \times (\text{EXPR} \times \text{VAL})$ is a set of flexible generic values, together with the corresponding inaccessible expression and its supposed type.

$$\begin{array}{c}
\text{INST-FLEX-L} \frac{k \vdash k' \mathbf{nocc} v}{k, \xi \vdash k' \mathbf{inst} v \Rightarrow (k', v)} k' \in \xi \\
\text{INST-FLEX-R} \frac{k \vdash k' \mathbf{nocc} v}{k, \xi \vdash v \mathbf{inst} k' \Rightarrow (k', v)} k' \in \xi \\
\text{INST-CON} \frac{k, \xi \vdash \vec{v} \mathbf{inst}_{\text{list}} \vec{w} \Rightarrow \sigma}{k, \xi \vdash \mathbf{C} \vec{v} \mathbf{inst} \mathbf{C} \vec{w} \Rightarrow \sigma} \quad \text{INST-DATA} \frac{k, \xi \vdash \vec{v} \mathbf{inst}_{\text{list}} \vec{w} \Rightarrow \sigma}{k, \xi \vdash \mathbf{D} \vec{v} \mathbf{inst} \mathbf{D} \vec{w} \Rightarrow \sigma} \\
\text{INST-EQ} \frac{k; \mathbf{N} \vdash v_1 \Leftrightarrow v_2}{k, \xi \vdash v_1 \mathbf{inst} v_2 \Rightarrow \diamond} \\
\frac{}{k, \xi \vdash \diamond \mathbf{inst}_{\text{list}} \diamond \Rightarrow \diamond} \quad \frac{k, \xi \vdash v_1 \mathbf{inst} w_1 \Rightarrow \sigma \quad k, \xi \vdash \sigma\{v_2\} \dots \sigma\{v_n\} \mathbf{inst}_{\text{list}} \sigma\{w_2\} \dots \sigma\{w_n\} \Rightarrow \sigma'}{k, \xi \vdash v_1 v_2 \dots v_n \mathbf{inst}_{\text{list}} w_1 w_2 \dots w_n \Rightarrow \text{comp } \sigma \sigma'}
\end{array}$$

Figure 3.14: Instantiation of flexible values by unification

- σ is a substitution for the flexible generic values
- p is the pattern to check
- v is the remaining type of the function.
- $(k', \xi', \sigma', \rho', \Gamma'), v'$ as “output”, are updated versions of the above.

This is simultaneously defined with checking a list of patterns in figure 3.15. Unification is used in the rule `CHKP-CON` for checking a constructor pattern.

3.5.5 Checking inaccessible patterns

After the accessible part of the patterns has been checked, now the inaccessible patterns have to be checked. Checking an inaccessible pattern \underline{e} is described in figure 3.16: The expression may contain any pattern variable of the clause. It is checked that the value inferred during the first phase is equal to the expression e written down by the user.

3.5.6 Checking the whole declaration

Now, only the right hand side is left to be checked. It is type-checked against the type remaining from the first phase. The complete algorithm for type-checking a mutual function declaration follows:

$$\begin{array}{c}
\text{CHKP-VAR} \frac{v_B = \searrow B^{\rho, x=k} \quad \rho' = (\rho, y = k) \quad \Gamma' = (\Gamma, y = v_A)}{(k, \xi, \sigma, \rho, \Gamma) \vdash y \text{ chkP Pi } x v_A B^\rho \Rightarrow (k+1, \xi, \sigma, \rho', \Gamma'), v_B} \\
\\
\text{CHKP-CON} \frac{\begin{array}{l} \Sigma \mathbf{c} = v_c \quad (k, \xi, \sigma, \rho, \Gamma) \vdash \vec{p} \text{ chkPs } v_c \Rightarrow (k', \xi', \sigma', \rho', \Gamma'), v'_c \\ k, \xi \vdash v'_c \text{ inst } v_A \Rightarrow \sigma_2 \quad v_p = \Downarrow^k p \quad \sigma'' = \text{comp } \sigma' \sigma_2 \\ \Gamma'' = \sigma''\{\Gamma'\} \quad v_B = \sigma''\{\searrow B^{\rho, x=v_p}\} \end{array}}{(k, \xi, \sigma, \rho, \Gamma) \vdash \mathbf{c} \vec{p} \text{ chkP Pi } x v_A B^\rho \Rightarrow (k', \xi', \sigma'', \rho', \Gamma''), v_B} \\
\\
\text{CHKP-INACC} \frac{v_B = \searrow B^{\rho, x=k}, \xi' = \xi \cup \{(k, (e, v_A))\}}{(k, \xi, \sigma, \rho, \Gamma) \vdash \underline{e} \text{ chkP Pi } x v_A B^\rho \Rightarrow (k+1, \xi', \sigma, \rho, \Gamma), v_B} \\
\\
\frac{}{(k, \xi, \sigma, \rho, \Gamma) \vdash \diamond \text{ chkPs } v \Rightarrow (k, \xi, \sigma, \rho, \Gamma), v} \\
\\
\frac{\begin{array}{l} (k, \xi, \sigma, \rho, \Gamma) \vdash p \text{ chkP } v \Rightarrow (k', \xi', \sigma', \rho', \Gamma'), v' \\ (k', \xi', \sigma', \rho', \Gamma') \vdash \vec{p} \text{ chkPs } v' \Rightarrow (k'', \xi'', \sigma'', \rho'', \Gamma''), v'' \end{array}}{(k, \xi, \sigma, \rho, \Gamma) \vdash p \vec{p} \text{ chkPs } v \Rightarrow (k'', \xi'', \sigma'', \rho'', \Gamma''), v''}
\end{array}$$

Figure 3.15: checking accessible patterns

$$\frac{\begin{array}{l} k; \rho; \Gamma \vdash e \Leftarrow \sigma\{v_i\} \\ (i, v_i) \in \sigma \quad k; \mathbf{N} \vdash \searrow e^\rho \Leftrightarrow v_i \end{array}}{(k, \sigma, \rho, \Gamma) \vdash i \text{ checkinacc } e : v_t}$$

Figure 3.16: Checking inaccessible pattern

Algorithm

For a mutual declaration $\delta =$

```
mutual
  fun  $f_1 : A_1$ 
     $\vec{\gamma}_1$ 
  ...
  fun  $f_n : A_n$ 
     $\vec{\gamma}_n$ 
```

and a type-correct signature Σ , if

1. for every $i \in \{1 \dots n\}$
 - (a) $1; \diamond; \diamond \vdash A_i$ **Type**
 - (b) $v_i := \searrow A_i^\diamond$
2. $\Sigma' := \Sigma \cup \bigcup_{i \in \{1 \dots n\}} \{f_i \mapsto (v_i, \vec{\gamma}_i, \perp)\}$
3. Using Σ' , for every $i \in \{1 \dots n\}$ and every clause $f \vec{p} e \in \vec{\gamma}_i$:
 - (a) $(1, \diamond, \diamond, \diamond, \diamond) \vdash \vec{p} \mathbf{chkPs} v_i \Rightarrow (k, \xi, \sigma, \rho, \Gamma), v$
 - (b) $(k, \sigma, \rho, \Gamma) \vdash i \mathbf{checkinacc} e : v_t$ for all $(i, (e, v_t)) \in \xi$
 - (c) $k; \rho; \Gamma \vdash e \Leftarrow v$
4. $\Sigma'' := \Sigma \cup \bigcup_{i \in \{1 \dots n\}} \{f_i \mapsto (v_i, \vec{\gamma}_i, \top)\}$

then Σ'' is a type-correct signature.

A mutual cofun declaration is type-checked just the same.

3.6 Mugda programs

The previous sections detailed algorithms for all declarations of Mugda. We can apply those to a list of declarations:

Algorithm

For a Mugda program $\vec{\delta} = \delta_1 \dots \delta_n$, if

- for every $i \in \{1 \dots n\}$: Σ_i is the result of type-checking δ_i using Σ_{i-1}

then $\vec{\delta}$ is a type-correct Mugda program.

So for a whole Mugda program, the declarations are checked one by one, starting with the empty signature Σ_0 .

Chapter 4

Termination-Checking

4.1 Motivation

The type-checking algorithm presented in the previous chapter has one drawback: The function declaration

```
fun foo : Nat → Nat
  foo x = foo x
```

passes the type checker, although the function denoted by `foo` is not well-defined. Also, now the declaration

```
let v : Vec Nat (foo zero) = nil Nat zero
```

is neither rejected nor accepted, because the evaluation of `foo zero` is undefined. For an implementation, this means that the type checker itself never finishes, or perhaps crashes with a stack overflow. Type-checking is not *decidable*, because it is not enforced that evaluation is well-defined.

Terminating closure

A closure e^ρ is *terminating* if its evaluation is defined, i.e. there exists a value $v \in \text{VAL}$ such that $\Downarrow e^\rho = v$.

Terminating signature

A type-correct signature Σ is *terminating*, if every closure that can be evaluated during type-checking with Σ is terminating.

Proposition: decidable type-checking

Type-checking of a Mugda program $\delta_1 \dots \delta_n$ is decidable if all signatures Σ_i that arise during type-checking (see section 3.6) are terminating.

Proposition

- The empty signature Σ_0 is terminating.
- Given
 - δ either a let , data or codata declaration
 - Σ a terminating signature
 - Σ' the signature resulting from type-checking δ with Σ .

then Σ' is terminating.

Only mutual (co)recursive declarations can possibly cause evaluation to be undefined. What is left to do in this chapter is to give a criterion that rejects all those declarations that would result in a signature that is not terminating. Of course, any such criterion will also reject valid ones, as the *halteproblem* is undecidable.

4.2 Matrix notation

For a set R we write $R^{m,n}$ for the set of matrices with m rows and n columns and elements in R . We write $\alpha_{i,j}$ for the element in the i th row and j th column. For a matrix $\alpha \in R^{m,n}$, the dimension is $|\alpha| := (m, n)$. For a square matrix $\beta \in R^{m,m}$, the set of diagonal elements is $\text{diag } \beta := \{\beta_{i,i} \mid i \leq m\}$.

4.3 Relating pattern and expressions

The size-change principle [LJBA01] states:

a program terminates on all inputs if every infinite call sequence would cause an infinite descent in some data values.

So to see that an evaluation in our semantics is terminating, we need to know how the size of the *semantical* values change during evaluation of recursion. This is done by analyzing the *syntax* of the program: how do the expression of the arguments from a recursive call relate to the patterns on the right hand side ?

This section is mostly based on [AA02], but where their work constructs a lexicographic order to show termination, we will use a simpler and yet more powerful criterion based on the size change principle, which is presented in the following section.

Order

$$\text{ORDER} = \{?, \leq, <\}$$

This set denotes the possible results when comparing an expression e to a pattern p , with the intended meaning:

- $e < p$: the value represented by e is *smaller* than that of p .
- $e \leq p$: we do not have $e_1 < p$, but the value represented by p is not smaller than that of e .
- $e ? p$: the value represented by p is smaller than e or their relation is not known.

Our order will be based on two rules:

- **Axiom 1:** $x < \mathbf{c} \vec{p}_1 x \vec{p}_n$ when \mathbf{c} is an inductive constructor
- **Axiom 2:** $f \vec{e} \leq f$

The first rule expresses that x is a structural part of $\mathbf{c} \vec{p}_1 x \vec{p}_n$. The second rule is essential for higher order data types like `Ord`.

Order multiplication

can be seen as serial composition:

*	<	≤	?
<	<	<	?
≤	<	≤	?
?	?	?	?

Order Addition

can be seen as parallel composition, or as the maximum:

+	<	≤	?
<	<	<	<
≤	<	≤	≤
?	<	≤	?

The triple $(\text{ORDER}, +, *)$ forms a commutative semiring, where $?$ is the neutral element of $+$ and \leq is the neutral element of $*$.

Order minimum

∧	<	≤	?
<	<	≤	?
≤	≤	≤	?
?	?	?	?

Maximum and minimum for list

for a non-empty list $(o_1 \dots o_n)$ we define as abbreviations

$$\max_o(o_1 \dots o_n) := o_1 + \dots + o_n$$

and

$$\min_o(o_1 \dots o_n) := o_1 \wedge \dots \wedge o_n$$

Arity

All clauses of a function declaration should have the same number of patterns.

The *arity* of a function f is $\text{ar}(f) := |\vec{p}|$ where $f \vec{p} = e$ is some clause of f .

Call

A *call* is an expression of the form $f \vec{e}$.

Expression to pattern

As said, we want to compare an expression to a pattern, yielding an element of ORDER. Mugda mainly differs from simply typed languages by having the inaccessible patterns \underline{e} . We handle this kind of pattern by trying to convert the expression e into a “normal” pattern:

$$\begin{aligned} \text{etp} : \text{EXPR} &\rightarrow \text{PAT} \cup \{ \uparrow \} \\ \text{etp } x &= x \\ \text{etp } (\mathbf{c } e_1 \dots e_n) &= \mathbf{c } (\text{etp } e_1) \dots (\text{etp } e_n) \\ \text{etp } \mathbf{c} &= \mathbf{c} \diamond \\ \text{etp } e &= \uparrow \text{ otherwise} \end{aligned}$$

Now the comparison of an expression to a pattern follows:

Comparing expression to pattern

$$\begin{aligned} \text{cmp} : \text{EXPR} \times \text{PAT} &\rightarrow \text{ORDER} \\ \text{cmp } e_1 \underline{e_2} &= \begin{cases} \text{cmp } e_1 p & \text{if } \text{etp } e_2 = p \neq \uparrow \\ \leq & \text{if } e_1 = e_2 \\ ? & \text{otherwise} \end{cases} \\ \text{cmp } x p &= \text{cmp}_v x p \\ \text{cmp } (x \vec{e}) p &= \text{cmp}_v x p \\ \text{cmp } \mathbf{c } (\mathbf{c} \diamond) &= \leq \\ \text{cmp } (\mathbf{c } e_1 \dots e_n) (\mathbf{c } p_1 \dots p_n) &= \min_o (\text{cmp } e_1 p_1) \dots (\text{cmp } e_n p_n) \\ \text{cmp } e p &= ? \text{ otherwise} \end{aligned}$$

where cmp_v compares a variable to a pattern:

Comparing variable to pattern

$$\begin{aligned}
\text{cmp}_v &: \mathbb{V} \times \text{PAT} \rightarrow \text{ORDER} \\
\text{cmp}_v x x &= \leq \\
\text{cmp}_v x (\mathbf{c} p_1 \dots p_n) &= < * \max_o (\text{cmp}_v x p_1) \dots (\text{cmp}_v x p_n) \\
&\quad \text{if } \mathbf{c} \text{ is inductive constructor} \\
\text{cmp}_v x \underline{e} &= \text{cmp}_v x p \text{ if } \text{etp } e = p \neq \uparrow \\
\text{cmp}_v x p &= ? \text{ otherwise}
\end{aligned}$$

Note that coinductive objects are not well-founded, so we have

$$\text{cmp}_v x (\mathbf{c} \vec{p}) = ?$$

when \mathbf{c} is a coinductive constructor.

Call matrices

A *call matrix* is a triple (f, α, g) where $f, g \in \mathbb{F}$ and $\alpha \in \text{ORDER}^{\text{ar}(f), \text{ar}(g)}$. It represents a call from f to \gg in the static flowgraph. The set of all call matrices is CALL.

The size-change principle is often presented with bipartite graphs instead of call matrices, but these are equivalent representations.

Call matrix of call

Let $f \vec{p} = e$ be a function clause and $g \vec{e}$ be a call in e , where $\vec{p} = p_1 \dots p_n$ and $\vec{e} = e_1 \dots e_m$. The associated call matrix $\text{cm}(f, \vec{p}, g, \vec{e}) \in \text{CALL}$ is given by

$$\text{cm}(f, \vec{p}, g, \vec{e}) := (f, \alpha, g)$$

where $\alpha \in \text{ORDER}^{\text{ar}(f), \text{ar}(g)}$ with the elements:

$$\alpha_{i,j} = \begin{cases} \text{cmp } e_j p_i & \text{if } j \leq m \\ ? & \text{if } j > m \end{cases}$$

In the definition of α , it was taken into consideration that there can be calls $f e_1 \dots e_n$ where $n < \text{arf}$, because a function might be used as a higher-order argument. The missing arguments are filled up with ?.

Call set

A finite set of call matrices is called a *call set*. For a mutual declaration

$$\delta = \text{mutual } (f_1 A_1 \gamma_1) \dots (f_n A_n \gamma_n)$$

we define the set *rec* of recursive function identifiers

$$\text{rec} := \{f_1, \dots, f_n\}$$

and the *initial call set* $cs(\delta)$ as

$$cs(\delta) := \bigcup_{i \in \{1 \dots n\}} \{\text{extr } rec \ f_i \vec{p} e \mid f_i \vec{p} e \in \gamma_i\}$$

The initial call set includes the call matrices of all recursive calls, where the extraction of call matrices from a right hand side is defined as follows:

Extraction of call matrices from expression

$$\begin{aligned} \text{extr} : \mathcal{P}(\mathbb{F}) \times \mathbb{F} \times \text{PAT}^* \times \text{EXPR} &\rightarrow \mathcal{P}(\text{CALL}) \\ \text{extr } rec \ f \vec{p} \ (\mathbf{g} \ e_1 \dots e_n) &= \{\text{cm}(f, \vec{p}, \mathbf{g}, \vec{e})\} \cup \bigcup_{i \in \{1 \dots n\}} \text{extr } rec \ f \vec{p} e_i \\ &\quad \text{if } \mathbf{g} \in \text{rec} \\ \text{extr } rec \ f \vec{p} \ \mathbf{g} &= \{\text{cm}(f, \vec{p}, \mathbf{g}, \diamond)\} \\ &\quad \text{if } \mathbf{g} \in \text{rec} \\ \text{extr } rec \ f \vec{p} \ (e \ e_1 \dots e_n) &= \text{extr } rec \ f \vec{p} e \cup \bigcup_{i \in \{1 \dots n\}} \text{extr } rec \ f \vec{p} e_i \\ \text{extr } rec \ f \vec{p} \ \lambda x. e &= \text{extr } rec \ f \vec{p} e \\ \text{extr } rec \ f \vec{p} \ \text{let } x : A = e_1 \text{ in } e_2 &= \text{extr } rec \ f \vec{p} e_1 \cup \text{extr } rec \ f \vec{p} e_2 \\ \text{extr } rec \ f \vec{p} \ (x : A) \rightarrow B &= \text{extr } rec \ f \vec{p} A \cup \text{extr } rec \ f \vec{p} B \\ \text{extr } rec \ f \vec{p} \ e &= \emptyset \text{ otherwise} \end{aligned}$$

Note that for the expression $\text{let } x : A = e_1 \text{ in } e_2$ we don't have to extract calls in the type A because it is only evaluated during type-checking of the function.

4.4 Applying the size-change principle

Starting from the initial call set, the following allows reasoning about the possibility of infinite call sequences:

Order matrix multiplication

$$\begin{aligned} \times : O^{m,n} \times O^{n,p} &\rightarrow O^{m,p} \\ (\alpha \times \beta)_{i,j} &= \sum_{k=1}^n \alpha_{i,k} * \beta_{k,j} \end{aligned}$$

Call Matrix composition

$$(f, \alpha, g) \star (g, \beta, h) := (f, \alpha \times \beta, h)$$

Call set completion

Completing a call set is closing it under composition. This can be achieved with a fixed point algorithm:

$$\begin{aligned} \text{complete} &: \mathcal{P}(\text{CALL}) \rightarrow \mathcal{P}(\text{CALL}) \\ \text{complete } cs &= \begin{cases} \text{complete } cs' & \text{if } cs \neq cs' \\ cs' & \text{if } cs = cs' \end{cases} \\ &\text{where } cs' = cs \cup \{(f, \alpha, g) \star (g', \beta, h) \mid g = g' \text{ and } (f, \alpha, g), (g', \beta, h) \in cs\} \end{aligned}$$

For a declaration δ , $\text{complete } cs(\delta)$ is the *completed call set* of δ .

Idempotent call matrix

A call-matrix (f, α, g) is *idempotent* if $f = g$ and $\alpha \times \alpha = \alpha$

Decreasing call matrix

The element $\langle \in O$ is *decreasing*.

A call matrix (f, α, g) is *decreasing* if $\text{diag}(\alpha)$ contains a decreasing element.

Proposition: Size-change principle for Mugda

We can now give a criterion based on the size-change principle [LJBA01]:

Given

1. a terminating signature Σ
2. a mutual recursive function declaration δ
3. Σ' is the signature resulting from type-checking δ in Σ
4. every idempotent call-matrix $\alpha \in \text{complete } (cs(\delta))$ is decreasing.

then Σ' is terminating.

The size-change principle is proven to be correct in [Wah07] for a dependently typed language with simpler *first order data types*. The work in [AA02] contains a proof of correctness for the structural order defined by axioms 1 and 2 in a simply typed language.

Productivity of corecursive declarations will follow in the next chapter, when the Size type is added to Mugda.

4.5 Examples

4.5.1 Addition

Recall the addition function on natural numbers:

```
fun add : Nat → Nat → Nat
  add x zero = x
  add x (succ y) = succ (add x y)
```

The single recursive call is `add x y` in the second clause. To build the call matrix we need to calculate:

```
cmp x x = ≤
cmp x (succ y) = ?
cmp y x = ?
cmp y (succ y) = <
```

The call set is a singleton set $\{\alpha\}$ where

$$\alpha = \text{add} \begin{pmatrix} \leq & ? \\ ? & < \end{pmatrix} \text{add}$$

To complete the call set, α is composed with itself:

$$\text{add} \begin{pmatrix} \leq & ? \\ ? & < \end{pmatrix} \text{add} \star \text{add} \begin{pmatrix} \leq & ? \\ ? & < \end{pmatrix} \text{add} = \text{add} \begin{pmatrix} \leq & ? \\ ? & < \end{pmatrix} \text{add}$$

which yields again α . Thus, $\{\alpha\}$ is the completed call set. α is an idempotent matrix, and because it is decreasing, the declaration for `add` is accepted.

Now suppose the arguments in the recursive call are *permuted*:

```
fun add2 : Nat → Nat → Nat
  add2 x zero = x
  add2 x (succ y) = succ (add2 y x)
```

which still computes the addition of two natural numbers. The initial call set is

$$\beta_1 = \text{add}_2 \begin{pmatrix} ? & \leq \\ < & ? \end{pmatrix} \text{add}_2$$

The completed call set is $\{\beta_1, \beta_2, \beta_3\}$ where

$$\beta_2 = \text{add}_2 \begin{pmatrix} < & ? \\ ? & < \end{pmatrix} \text{add}_2, \beta_3 = \text{add}_2 \begin{pmatrix} ? & < \\ < & ? \end{pmatrix} \text{add}_2$$

The only idempotent matrix β_2 is decreasing, so the definition of `add2` is also accepted. Permuted arguments are a strength of the size-change principle.

4.5.2 Mutual even and odd

The following mutual declaration

```
mutual
  fun even : Nat → Bool
    even zero = tt
    even (succ x) = odd x
  fun odd : Nat → Bool
    odd zero = ff
    odd (succ x) = even x
```

yields the initial call set

$$\{ \text{even } (<) \text{ odd}, \text{ odd } (<) \text{ even} \}$$

The completed call set

$$\{ \text{even } (<) \text{ odd}, \text{ odd } (<) \text{ even}, \text{ even } (<) \text{ even}, \text{ odd } (<) \text{ odd} \}$$

includes two additional idempotent matrices that are both decreasing. Thus this mutual definition is also accepted.

4.5.3 Brouwer ordinals

The so-called Brouwer ordinals can be defined with

```
data Ord : Set
  ozero : Ord
  olim : (Nat → Ord) → Ord
```

This is an example of a higher-order data type, because the argument to `olim` contains a function space. Now ordinal addition can be defined:

```
fun addOrd : Ord → Ord → Ord
  addOrd x ozero = x
  addOrd x (olim f) = olim (λ y. addOrd x (f y))
```

The interesting comparison is $(f y)$ against $\text{olim } f$. We should have $f y \leq f$ (Axiom 2) and $\text{olim } f < f$ (Axiom 1), thus with transitivity $f y < \text{olim } f$. And indeed we get:

```
cmp x x = ≤
cmp x (olim f) = ?
cmp (f y) x = ?
cmp (f y) (olim f) = cmpv f (olim f) = < * maxo (cmpv f f) = < * ≤ = <
```

The corresponding call set is

$$\text{addOrd} \begin{pmatrix} \leq & ? \\ ? & < \end{pmatrix} \text{addOrd}$$

and is then accepted just like `add`.

4.6 Extending the order

Motivation

The following type

```
data NatPair : Set
  np : Nat → Nat → NatPair
```

denotes pairs of natural numbers. Now if one defines addition on pairs:

```
fun add_p : NatPair → Nat
  add_p (np x zero) = x
  add_p (np x (succ y)) = succ (add_p (np x y))
```

but this does no longer pass the termination-check. Too much information is lost when creating the call matrices for this single nested pattern. For add_p , the call matrix is

$$\text{add}_p (\leq) \text{add}_p$$

which is idempotent but not decreasing.

The plan is to do a finer comparison for nested patterns, where the result is not a single element, but a whole matrix of elements:

Extended Order

The set ORDER_+ is defined recursively by

- $\text{ORDER} \subseteq \text{ORDER}_+$
- if $M \subseteq \text{ORDER}_+$ and $n \in \mathbb{N}$ then $M^{n,n} \subseteq \text{ORDER}_+$

Extended Order operations

The operations $*$, $+$ and \wedge given in the previous section are extended to ORDER_+ by adding the following additional cases together with a new operation, collapse :

Additional composition cases

$$\begin{aligned} \alpha * \beta &= \begin{cases} \alpha \times \beta & \text{if } |\alpha| = |\beta| \\ (\text{collapse } \alpha) * (\text{collapse } \beta) & \text{otherwise} \end{cases} \\ \alpha * ? &= ? \\ \alpha * \leq &= \alpha \\ \alpha * < &= (\text{collapse } \alpha) * < \\ ? * \beta &= ? \\ \leq * \beta &= \beta \\ < * \beta &= < * (\text{collapse } \beta) \end{aligned}$$

Additional addition cases

$$\begin{aligned}
\alpha + \beta &= \begin{cases} \gamma \text{ with } \gamma_{i,j} = \alpha_{i,j} + \beta_{i,j} & \text{if } |\alpha| = |\beta| \\ (\text{collapse } \alpha) + (\text{collapse } \beta) & \text{otherwise} \end{cases} \\
\alpha + ? &= \alpha \\
\alpha + \leq &= (\text{collapse } \alpha) + \leq \\
\alpha + < &= < \\
? + \beta &= \beta \\
\leq + \beta &= \leq + (\text{collapse } \beta) \\
< + \beta &= <
\end{aligned}$$

Additional minimum cases

$$\begin{aligned}
\alpha \wedge \beta &= \begin{cases} \gamma \text{ with } \gamma_{i,j} = \alpha_{i,j} \wedge \beta_{i,j} & \text{if } |\alpha| = |\beta| \\ (\text{collapse } \alpha) \wedge (\text{collapse } \beta) & \text{otherwise} \end{cases} \\
\alpha \wedge ? &= ? \\
\alpha \wedge \leq &= (\text{collapse } \alpha) \wedge \leq \\
\alpha \wedge < &= \alpha \\
? \wedge \beta &= ? \\
\leq \wedge \beta &= \leq \wedge (\text{collapse } \beta) \\
< \wedge \beta &= \beta
\end{aligned}$$

Collapsing of matrix

$$\begin{aligned}
\text{collapse} &: M(\text{ORDER}_+) \rightarrow \text{ORDER}_+ \\
\text{collapse } \alpha &= \min_o(\text{diag } \alpha)
\end{aligned}$$

Comparison with extended order

Now, the following clause of `cmp`

$$\text{cmp } (\mathbf{c} e_1 \dots e_n) (\mathbf{c} p_1 \dots p_n) = \min_o(\text{cmp } e_1 p_1) \dots (\text{cmp } e_n p_n)$$

is changed to

$$\text{cmp } (\mathbf{c} e_1 \dots e_n) (\mathbf{c} p_1 \dots p_n) = \begin{cases} \gamma \in \text{ORDER}_+^{n,n} \text{ with } \gamma_{i,j} = \text{cmp } e_j p_i \\ \text{if } n \geq 2 \\ \text{cmp } e_1 p_1 \text{ otherwise} \end{cases}$$

so more order information about the components is used, for as long as possible before the matrix might have to be collapsed.

Decreasing element

In the extended order, a decreasing element is recursively defined by

- $<$ is decreasing
- α is decreasing if $\text{diag}(\alpha)$ has at least one decreasing element.

The remaining operations from the previous section carry over directly to ORDER_+ . Before giving examples, we conjecture the following:

Proposition: Size-change principle for extended order

The size change principle with ORDER_+ for Mugda holds.

4.6.1 Examples**Pair addition**

Back to the example from the motivation, the initial call set calculated with the extended order is:

$$\text{add}_p \left(\begin{pmatrix} \leq & ? \\ ? & < \end{pmatrix} \right) \text{add}_p$$

This is also the completed call set and the matrix is decreasing.

List flattening

Another example that now is recognized as terminating is this version of list flattening:

```
fun flat : (A : Set) → List (List A) → List A
  flat A (nil List A) = nil A
  flat A (cons List A (nil A) yl) = flat A yl
  flat A (cons List A (cons A x xl) yl) = cons A x (flat A (cons (List A) xl yl))
```

The initial set of call matrices is:

$$\text{flat} \begin{pmatrix} \leq & < \\ ? & < \end{pmatrix} \text{flat}, \text{flat} \begin{pmatrix} \leq & < \\ ? & \begin{pmatrix} \leq & ? & ? \\ ? & < & ? \\ ? & ? & \leq \end{pmatrix} \end{pmatrix} \text{flat}$$

It turns out that this is already the completed call set. Both matrices are idempotent and decreasing.

4.7 List reversion: Vectors to the rescue

First, we present a peculiar list reversion algorithm found in [Bla04]. It is shown in figure 4.1. This mutual declaration is not accepted as terminating. For example the call

$$\text{rev } A \text{ (rev}_2 \text{ } A \text{ } x \text{ } xs)$$

in the third clause of rev_2 induces the following call matrix:

$$\text{rev}_2 \begin{pmatrix} \leq & ? \\ \leq & ? \end{pmatrix} \text{rev}$$

Indirect calls are not handled well by a purely syntactic test. A termination-checker needs information about the behaviour of rev_2 , more precisely about the size of the returned list. But the user can help by using vectors instead of lists: the same algorithm with vectors is shown in figure 4.2. This declaration is now accepted, because in enough recursive calls the first argument is getting structurally smaller.

mutual

```

fun rev : (A : Set) → List A → List A
  rev A (nil A) = nil A
  rev A (cons A x xs) = cons A (rev1 A x xs) (rev2 A x xs)
fun rev1 : (A : Set) → A → List A → A
  rev1 A a (nil A) = a
  rev1 A a (cons A x xs) = rev1 A x xs
fun rev2 : (A : Set) → A → List A → List A
  rev2 A a (nil A) = nil A
  rev2 A a (cons A x xs) = rev A (cons A a (rev A (rev2 A x xs)))

```

Figure 4.1: not accepted : reversion on Lists

```

mutual
fun rev : (n : Nat) → (A : Set) → Vec A n → Vec A n
  rev zero A (nil A) = nil A
  rev succ n A (cons A n x xs) =
    cons A n (rev1 n A x xs) (rev2 n A x xs)
fun rev1 : (n : Nat) → (A : Set) → A → Vec A n → A
  rev1 zero A a (nil A) = a
  rev1 succ n A a (cons A n x xs) = rev1 n A x xs
fun rev2 : (n : Nat) → (A : Set) → A → Vec A n → Vec A n
  rev2 zero A a (nil A) = nil A
  rev2 succ n A a (cons A n x xs) =
    rev (succ n) A (cons A n a (rev n A (rev2 n A x xs)))

```

Figure 4.2: accepted: reversion on vectors

Chapter 5

Sized data types

As the `rev` example demonstrated, richer types allow more declarations to be accepted by the termination-checker. But natural numbers are not sufficient as the height for inhabitants of coinductive types like `Stream` or types with infinite branching like `Ord`. This is why a type `Size` will be added to `Mugda` that has an element ∞ that is “big enough” for all objects in `Mugda`. Inductive families that use this new type as an index will be called *sized types*.

In addition, using vectors instead of lists is not always easily possible. Consider a `filter` function that removes some elements from a list. For vectors, we would need to resort to existential types¹, which we will sketch as

$$\begin{aligned} \text{fun filter} : (A : \text{Set}) \rightarrow (p : A \rightarrow \text{Bool}) \rightarrow (n : \text{Nat}) \\ \rightarrow \text{Vec } A \ n \rightarrow (m : \text{Nat} . (\text{Leq } m \ n , \text{Vec } A \ m)) \end{aligned}$$

where `filter` returns a natural number m , a proof that m is smaller than n , and a vector of length m .

The `Size` type will provide an easy way out. For inductive types, the size will be an upper bound on the height. Filtering on sized lists then has the type

$$\text{fun filter} : (A : \text{Set}) \rightarrow (p : A \rightarrow \text{Bool}) \rightarrow (i : \text{Size}) \rightarrow \text{List } A \ i \rightarrow \text{List } A \ i$$

and `Mugda` will offer subtyping that is helpful when implementing `filter`.

5.1 Adding a `Size` type

In most works on sized types, sizes annotations are merely “tagged on” to an existing language. An exception is [Abe06], where a *size kind* is added to a polymorphic lambda calculus. As `Mugda` already supports inductive families, adding a primitive type of sizes was the natural choice.

¹for the use of existential types for termination, see [XP99].

5.1.1 Syntax and semantics

We extend expressions and patterns:

$$\begin{aligned} \text{EXPR } \ni e, A, B & ::= \dots \\ & \quad | \text{ Size } \quad \text{Size type} \\ & \quad | \mathbf{s} e \quad \text{size successor} \\ & \quad | \infty \quad \text{limit size} \end{aligned}$$

$$\begin{aligned} \text{PAT } \ni p & ::= \dots \\ & \quad | \mathbf{s} p \quad \text{size successor pattern} \end{aligned}$$

As `Size` is a type, in particular we also have variables of type `Size`. Thus, we can form size expressions of the form that was first developed in [BFG⁺04].

Semantics

Values also are extended accordingly:

$$\begin{aligned} \text{VAL } \ni v & ::= \dots \\ & \quad | \text{ Size } \quad \text{Size type} \\ & \quad | \mathbf{s} v \quad \text{size successor} \\ & \quad | \infty \quad \text{size limit} \end{aligned}$$

We will now expand semantics for the size type. For the semantics, the size type can be imagined as a coinductive type with one constructor `s` and a distinguished inhabitant `∞`.

Size successor

The semantics should take into account that `∞` is the limit size, i.e. the equation `s ∞ = ∞` holds:

$$\begin{aligned} \mathbf{s}_\infty & : \text{VAL} \rightarrow \text{VAL} \\ \mathbf{s}_\infty \infty & = \infty \\ \mathbf{s}_\infty v & = \mathbf{s} v \text{ otherwise} \end{aligned}$$

Evaluation

$$\begin{aligned} \searrow \text{Size}^\rho & = \text{Size} \\ \searrow \infty^\rho & = \infty \\ \searrow (\mathbf{s} e)^\rho & = \mathbf{s}_\infty v \text{ where } v = \searrow e^\rho \end{aligned}$$

Pattern matching

$$\begin{aligned} \text{match}_f \rho (\mathbf{s} p) \infty & = \text{match } \rho p \infty \\ \text{match}_f \rho (\mathbf{s} p) (\mathbf{s} v) & = \text{match } \rho p v \end{aligned}$$

5.1.2 Type-Checking

First the following clauses are added:

Application of substitution

$$\sigma\{\mathbf{s} v\} = \mathbf{s}_\infty \sigma\{v\}$$

Converting pattern to value

$$\mathbf{p}2v k(\mathbf{s} p) = (\mathbf{s} v, k') \text{ where } (v, k') = \mathbf{p}2v k p$$

All additional typing rules are shown in figure 5.1. It is noteworthy that **Size** is not a small type. The rule **CHKP-SUCC** is similar to **CHKP-CON**, as **s** can be seen as a constructor of type $\text{Size} \rightarrow \text{Size}$.

$$\begin{array}{c} \frac{}{k; \rho; \Gamma \vdash \mathbf{Size} \ \mathbf{Type}} \quad \text{CHKP-SUCC} \frac{k; \rho; \Gamma \vdash e \Leftarrow \mathbf{Size}}{k; \rho; \Gamma \vdash \mathbf{s} e \Leftarrow \mathbf{Size}} \\ \\ \text{INF-INFY} \frac{}{k; \rho; \Gamma \vdash \infty \Rightarrow \mathbf{Size}} \\ \\ \text{EQ-SUCC} \frac{f; k \vdash v_1 \Leftrightarrow v_2}{f; k \vdash \mathbf{s} v_1 \Leftrightarrow \mathbf{s} v_2} \\ \\ \frac{k \vdash a \ \mathbf{no}cc \ v}{k \vdash a \ \mathbf{no}cc \ \mathbf{s} v} \quad \frac{k \vdash a \ \mathbf{s}pos \ v}{k \vdash a \ \mathbf{s}pos \ \mathbf{s} v} \\ \\ \text{CHKP-SUCC} \frac{(k, \xi, \sigma, \rho, \Gamma) \vdash p \ \mathbf{chkP} \ \mathbf{Size} \rightarrow \mathbf{Size} \Rightarrow (k', \xi', \sigma', \rho', \Gamma'), v' \quad v = \Downarrow^k p}{(k, \xi, \sigma, \rho, \Gamma) \vdash \mathbf{s} p \ \mathbf{chkP} \ \mathbf{Pi} \ x \ \mathbf{Size} \ B^\rho \Rightarrow (k', \xi', \sigma', \rho', \Gamma'), \searrow B^{\rho, x=v}} \\ \\ \text{INST-SUCC} \frac{k, \xi \vdash v_1 \ \mathbf{inst} \ v_2 \Rightarrow \sigma}{k, \xi \vdash \mathbf{s} v_1 \ \mathbf{inst} \ \mathbf{s} v_2 \Rightarrow \sigma} \\ \\ \text{INST-INFY} \frac{k, \xi \vdash v_1 \ \mathbf{inst} \ \infty \Rightarrow \sigma}{k, \xi \vdash \mathbf{s} v_1 \ \mathbf{inst} \ \infty \Rightarrow \sigma} \end{array}$$

Figure 5.1: Additional typing rules for **Size** type

5.1.3 Termination-Checking

We now detail the additions to chapter 4:

Expression to pattern

$$\text{etp } (\mathbf{s } e) = \mathbf{s } (\text{etp } e)$$

Comparing expression to pattern

$$\text{cmp } (\mathbf{s } e_1) (\mathbf{s } e_2) = \text{cmp } e_1 e_2$$

Comparing variable to pattern

the size successor is treated like an *inductive* constructor:

$$\text{cmp}_v x (\mathbf{s } p) = < * \text{cmp}_v x p$$

This is quite dangerous, as the `Size` type is not well-founded. Later the use of `Size` must be controlled so that termination-checking remains valid.

5.2 Sized data type declarations

Two new declarations are added to the language:

Declarations

$$\begin{aligned} \text{DECL } \ni \delta ::= & \dots \\ & | \text{ sized data } D \tau : A \vec{\gamma} \quad \text{sized inductive data type} \\ & | \text{ sized codata } D \tau : A \vec{\gamma} \quad \text{sized co inductive data type} \end{aligned}$$

5.2.1 Examples**Sized natural numbers**

```
sized data Nat : Size → Set
  zero : (i : Size) → Nat (s i)
  succ : (i : Size) → Nat i → Nat (s i)
```

The following declaration

```
let 1 : Nat ∞ = succ ∞ (zero ∞)
```

defines the natural number 1. So “at run-time” all objects of a sized type get the height ∞ , which is the only closed expression of type `Size`. We also introduce the type of sized streams:

Sized Streams

```
sized codata Stream : Size → Set
  cons : (i : Size) → Nat → Stream i → Stream (s i)
```

Next is type-checking of the new sized declarations.

5.2.2 Checking sized data type declarations

A sized data declaration need to be of the syntactic scheme outlined in figure 5.2:

$$\begin{aligned}
 & \text{sized data } D (p_1 : P_1) \dots (p_n : P_n) : \text{Size} \rightarrow \Theta \rightarrow \text{Set} \\
 & \mathbf{c}_1 : (i_1 : \text{Size}) \rightarrow \Delta_1 \rightarrow D p_1 \dots p_n (\mathbf{s} i_1) t_2^1 \dots t_m^1 \\
 & \dots \\
 & \mathbf{c}_k : (i_k : \text{Size}) \rightarrow \Delta_k \rightarrow D p_1 \dots p_n (\mathbf{s} i_k) t_2^k \dots t_m^k
 \end{aligned}$$

Figure 5.2: Syntactic valid form of sized data declaration

For non-sized data declarations, a size is not allowed in the indices Θ because **Size** is not a small type. But, for a sized declarations, it is allowed as the first index and, for every constructor, as the first argument. We will check that every recursive argument in every Δ_j has the form

$$D \dots i \dots$$

and that i does not occur anywhere else (figure 5.3). The judgement $k \vdash v \text{ sizeCon } i D$ is used to check every constructor, where

- k denotes the next free generic value.
- v is the value that remains to be checked.
- $i \in \mathbb{N}$ is the generic value of the size index.
- D is the declared sized data type.

The helper judgment $k \vdash v \text{ sizeUse } i D$ is used to check the occurrences of i in every constructor argument. The size argument should denote the height of an object when viewed as a tree, and every constructor increases this height.

In summary, the type-checking algorithm for sized data types follows:

Algorithm

For the declaration $\delta = \text{sized data } D \tau : A \vec{\gamma}$ and a type-correct signature Σ , if

1. $n := |\tau|$
2. δ follows the form of figure 5.2
3. $1; \diamond; \diamond; n + 1 \vdash \tau \rightarrow A \text{ DataType}$
4. $v_D := \searrow (\tau \rightarrow A)^\diamond$
5. $\Sigma' := \Sigma \cup \{D \mapsto (v_D, n)\}$

6. Given Σ' , for every constructor declaration $c : B \in \vec{\gamma}$:
- (a) $1; \diamond; \diamond; n + 1 \vdash \tau \rightarrow B \text{ ConType}$
 - (b) $v_i := \searrow (\tau \rightarrow B)^\diamond$
 - (c) $1 \vdash j \text{ sposc } v_i$ for every $j \in \text{pos}(D)$
 - (d) $1 \vdash D \text{ sposc } v_i$
 - (e) $1 \vdash v_i \text{ sizeCon } (n + 1) D$
7. $\Sigma'' := \Sigma' \cup \bigcup_i \{c_i \mapsto v_i\}$

then Σ'' is a type-correct signature.

$$\frac{k \vdash i \text{ nocc } t_j \text{ for all } j \in \{2 \dots m\}}{k \vdash D p_1 \dots p_n i t_2 \dots t_m \text{ sizeUse } i D} \quad \Sigma D = (v_D, n)$$

$$\frac{\begin{array}{c} k \vdash v \text{ sizeUse } i D \\ k \vdash v_j \text{ sizeUse } i D \text{ for all } j \in \{1 \dots n\} \end{array}}{k \vdash v v_1 \dots v_n \text{ sizeUse } i D}$$

$$\frac{k \vdash v \text{ sizeUse } i D}{k \vdash s v \text{ sizeUse } i D} \quad \frac{\begin{array}{c} k \vdash v_A \text{ sizeUse } i D \\ k + 1 \vdash \searrow B^{\rho, x=k} \text{ sizeUse } i D \end{array}}{k \vdash \text{Pi } x v_A B^\rho \text{ sizeUse } i D}$$

$$\frac{k + 1 \vdash \searrow e^{\rho, x=k} \text{ sizeUse } i D}{k \vdash \text{Lam } x e^\rho \text{ sizeUse } i D} \quad \frac{a \neq i}{k \vdash a \text{ sizeUse } i D}$$

$$\frac{\begin{array}{c} k \vdash v_A \text{ sizeUse } i D \text{ if } k > i \\ v_A = \text{Size} \text{ if } k = i \\ k + 1 \vdash \searrow B^{\rho, x=k} \text{ sizeCon } i D \end{array}}{k \vdash \text{Pi } x v_A B^\rho \text{ sizeCon } i D}$$

$$\frac{\begin{array}{c} k \vdash i \text{ nocc } p_j \text{ for all } j \in \{1 \dots n\} \\ k \vdash i \text{ nocc } t_j \text{ for all } j \in \{2 \dots m\} \end{array}}{k \vdash D p_1 \dots p_n (s i) t_2 \dots t_m \text{ sizeCon } i D} \quad \Sigma D = (v_D, n)$$

Figure 5.3: constructor size check

5.3 Subtyping for size

For an inductive sized type, the size index can be interpreted as an *upper bound* on the height of its inhabitants. An inhabitant of $\text{Nat } i$ is also an

$$\begin{array}{cc}
\text{SLEQ-INFTY} \frac{}{\vdash v \sqsubseteq \infty} & \text{SLEQ-GEN} \frac{}{\vdash k \sqsubseteq k} \\
\text{SLEQ-SUCC-I} \frac{\vdash v_1 \sqsubseteq v_2}{\vdash \mathbf{S} v_1 \sqsubseteq \mathbf{S} v_2} & \text{SLEQ-SUCC-II} \frac{\vdash v_1 \sqsubseteq v_2}{\vdash v_1 \sqsubseteq \mathbf{S} v_2}
\end{array}$$

Figure 5.4: Size value comparison

inhabitant of $\text{Nat } (s i)$. For an inductive sized types like Nat the following should hold:

$\text{Nat } i$ is a subtype of $\text{Nat } (s i)$

and

$\text{Nat } i$ is a subtype of $\text{Nat } \infty$.

for every size i .

For Stream as a sized coinductive type, the type $\text{Stream } \infty$ contains all fully defined streams. Thus, the size can be interpreted as a *lower bound* for the definedness of its inhabitants:

$\text{Stream } (s i)$ is a subtype of $\text{Stream } i$

and

$\text{Stream } \infty$ is a subtype of $\text{Stream } i$

for every size i .

A partial order on values of type Size is given in figure 5.4. As strict positivity implies positivity (monotonicity), the information about strictly positive parameters can be used for subtyping. The subtype relation in figure 5.5 is defined, just like the equality relation, by two simultaneous judgments

$$f; k \vdash v_1 \leq v_2 \subseteq \mathbb{F} \times \mathbb{N} \times \text{VAL} \times \text{VAL}$$

$$f; k \vdash v_1 \ll v_2 \subseteq \mathbb{F} \times \mathbb{N} \times \text{VAL} \times \text{VAL}$$

where again the first judgment is used to unroll one of the values v_1 and v_2 for the second judgment. Of special interest are the rules:

- LEQ-IND: subtyping for sized inductive type.
- LEQ-CO: subtyping for sized coinductive type.
- LEQ-DATA: subtyping for other data types.
- LEQ-PI: subtyping is *contravariant* for the function domain.

As the values “switch places” for the domain in the rule `LEQ-PI`, the force history f is switched accordingly. We define $\bar{\mathbf{R}} = \mathbf{L}$, $\bar{\mathbf{L}} = \mathbf{R}$ and $\bar{\mathbf{N}} = \mathbf{N}$.

Subtyping is now replacing type equality in the rule `CHK-INF`. The updated rule is shown in figure 5.6.

5.4 Examples: sized inductive types

5.4.1 Natural number division

The minus function can be defined by:

```
fun minus : (i : Size) → Nat i → Nat ∞ → Nat i
  minus si (zero i) y = zero i
  minus ix (zero ∞) = x
  minus si (succ i x) (succ ∞ y) = minus i x y
```

Subtyping is used in the right hand side of the last clause. While `minus` is structurally recursive even without a size, there is more information in the type: the size of the result list now has an upper bound i . This is essential for the following definition of division on natural numbers:

```
fun div : (i : Size) → Nat i → Nat ∞ → Nat i
  div si (zero i) y = zero i
  div si (succ i x) (zero ∞) = zero i
  div si (succ i x) (succ ∞ y) =
    succ i (div i (minus i x y) (succ ∞ y))
```

`div` is structural recursive on the size argument. The initial call set is the singleton matrix

$$\text{div} \begin{pmatrix} < & < & ? \\ ? & ? & ? \\ ? & ? & \leq \end{pmatrix} \text{div}$$

which is idempotent and decreasing.

$$\begin{array}{c}
\text{LEQ-FORCE} \frac{\begin{array}{l} \mathbf{L}; k \vdash \text{force } v_1 \ll v_2 \quad \text{if force } v_1 \neq v_1, \text{force } v_2 = v_2, f \neq \mathbf{R} \\ \mathbf{R}; k \vdash v_1 \ll \text{force } v_2 \quad \text{if force } v_1 = v_1, \text{force } v_2 \neq v_2, f \neq \mathbf{L} \\ f; k \vdash v_1 \ll v_2 \quad \text{otherwise} \end{array}}{f; k \vdash v_1 \leq v_2} \\
\\
\text{LEQ-IND} \frac{\begin{array}{l} \text{for all } j \in \text{pos}(\mathbf{D}) : f; k \vdash p_j \leq q_j \\ \text{for all } j \notin \text{pos}(\mathbf{D}) : f; k \vdash p_j \Leftrightarrow q_j \\ \text{for all } j \in \{1, \dots, m\} : f; k \vdash v_j \Leftrightarrow w_j \quad \vdash s \sqsubseteq t \end{array}}{f; k \vdash \mathbf{D} p_1 \dots p_n s v_1 \dots v_m \ll \mathbf{D} q_1 \dots q_n t w_1 \dots w_m} \Sigma \mathbf{D} = (v_{\mathbf{D}}, n) \\
\\
\text{LEQ-CO} \frac{\begin{array}{l} \text{for all } j \in \text{pos}(\mathbf{D}) : f; k \vdash p_j \leq q_j \\ \text{for all } j \notin \text{pos}(\mathbf{D}) : f; k \vdash p_j \Leftrightarrow q_j \\ \text{for all } j \in \{1, \dots, m\} : f; k \vdash v_j \Leftrightarrow w_j \quad \vdash t \sqsubseteq s \end{array}}{f; k \vdash \mathbf{D} p_1 \dots p_n s v_1 \dots v_m \ll \mathbf{D} q_1 \dots q_n t w_1 \dots w_m} \Sigma \mathbf{D} = (v_{\mathbf{D}}, n) \\
\\
\text{LEQ-DATA} \frac{\begin{array}{l} \text{for all } j \in \text{pos}(\mathbf{D}) : f; k \vdash p_j \leq q_j \\ \text{for all } j \notin \text{pos}(\mathbf{D}) : f; k \vdash p_j \Leftrightarrow q_j \\ \text{for all } j \in \{1, \dots, m\} : f; k \vdash v_j \Leftrightarrow w_j \end{array}}{f; k \vdash \mathbf{D} p_1 \dots p_n v_1 \dots v_m \ll \mathbf{D} q_1 \dots q_n w_1 \dots w_m} \Sigma \mathbf{D} = (v_{\mathbf{D}}, n) \\
\\
\text{LEQ-APP} \frac{f; k \vdash v \leq w \quad \text{for all } j \in \{1, \dots, m\} : f; k \vdash v_j \Leftrightarrow w_j}{f; k \vdash v v_1 \dots v_n \ll w w_1 \dots w_n} \\
\\
\text{LEQ-PI} \frac{\bar{f}; k \vdash v_2 \leq v_1 \quad f; k + 1 \vdash \searrow b_1^{\rho, x_1=k} \leq \searrow b_2^{\Gamma, x_2=k}}{f; k \vdash \text{Pi } x_1 v_1 b_1^{\rho} \ll \text{Pi } x_2 v_2 b_2^{\Gamma}} \\
\\
\text{LEQ-LAM} \frac{f; k + 1 \vdash \searrow e_1^{\rho, x_1=k} \leq \searrow e_2^{\Gamma, x_2=k}}{f; k \vdash \text{Lam } x_1 e_1^{\rho} \ll \text{Lam } x_2 e_2^{\Gamma}} \\
\\
\text{LEQ-SUCC} \frac{f; k \vdash v_1 \leq v_2}{f; k \vdash \mathbf{S} v_1 \ll \mathbf{S} v_2} \quad \text{LEQ-ATOM} \frac{}{f; k \vdash a \ll a}
\end{array}$$

Figure 5.5: subtype checking

$$\text{CHK-INF} \frac{k; \rho; \Gamma \vdash e \rightrightarrows v_2 \quad \mathbf{N}; k \vdash v_2 \leq v_1}{k; \rho; \Gamma \vdash e \Leftarrow v_1}$$

Figure 5.6: updated type-checking rule for subtyping

5.4.2 Sized Lists

```

sized data List (A : Set) : Size → Set
  nil : (i : Size) → List A (s i)
  cons : (i : Size) → A → List A i → List A (s i)

```

Quicksort

This version of the well-known sorting algorithm adapted from [Abe04] is shown in figure 5.7.

```

data Prod (+A : Set) : Set
  prod : A → A → Prod A
fun pr1 : (A : Set) → Prod A → A
  pr1 A (prod A a b) = a
fun pr2 : (A : Set) → Prod A → A
  pr2 A (prod A a b) = b
fun split : (i : Size) → (A : Set) → (leq : A → A → Bool)
  → A → List A i → Prod (List A i)
  split s i A leq a (nil A i) = prod (List A (s i)) (nil A i) (nil A i)
  split s i A leq a (cons A i x xs) =
    let rec : Prod (List A i) = split i A leq a xs in
    let l1 : List A i = pr1 (List A i) rec in
    let l2 : List A i = pr2 (List A i) rec in
    ite (Prod (List A (s i))) (leq a x)
      (prod (List A (s i)) l1 (cons A i x l2))
      (prod (List A (s i)) (cons A i x l1) l2)
fun qsapp : (i : Size) → (A : Set) → (leq : A → A → Bool)
  → List A i → List A ∞ → List A ∞
  qsapp s i A leq (nil A i) ys = ys
  qsapp s i A leq (cons A i x xs) ys =
    let sl : Prod (List A i) = split i A leq x xs in
    let l1 : List A i = pr1 (List A i) sl in
    let l2 : List A i = pr2 (List A i) sl in
    qsapp i A leq l1 (cons A ∞ x (qsapp i A leq l2 ys))
let quicksort : (i : Size) → (A : Set) → (leq : A → A → Bool)
  → List A i → List A ∞
  = λ i. λ A. λ leq. λ l. qsapp i A leq l (nil A ∞)

```

Figure 5.7: Quicksort

The main routine `quicksort` is parameterized over a linear order $leq : A \rightarrow A \rightarrow \text{Bool}$ on A . The `split` function splits a list l into a pair of list l_1 and l_2 where all smaller elements are in l_1 and the others in l_2 . The type of this

function

$$(i : \text{Size}) \rightarrow \dots \rightarrow \text{List } A \ i \rightarrow \text{Prod } (\text{List } A \ i)$$

allows a rough upper bound on the sizes of both result lists. This example shows how little effort sized types require to make some definitions accepted to the termination check, but the price to pay is that the type of quicksort itself is

$$(i : \text{Size}) \rightarrow \dots \rightarrow \text{List } A \ i \rightarrow \text{List } A \ \infty$$

so the information that quicksort is size preserving is lost.

5.4.3 Sized Brouwer ordinals

Now sized Brouwer ordinals are introduced:

```
sized data Ord : SizeSet
  ozero : (i : Size) → Ord (s i)
  olim  : (i : Size) → (Nat → Ord i) → Ord (s i)
```

With sized ordinals, Axiom 2 ($f \vec{e} \leq f$) is not needed to accept ordinal addition:

```
fun addOrd : Ord ∞ → (i : Size) → Ord i → Ord ∞
  addOrd x si (ozero i) = x
  addOrd x si (olim f) = olim ∞ (λ y. addOrd x i (f y))
```

addOrd is structurally recursive in the size argument.

5.4.4 A higher-order function

While somewhat artificial, the following example is interesting for two reasons:

```
fun addWith : ((k : Size) → Nat k → Nat k) → (i : Size) → (j : Size)
  → Nat i → Nat j → Nat ∞
  addWith f si j (zero i) y = y
  addWith f si j (succ i x) y = succ ∞ (addWith f j i y (f i x))
```

addWith resembles the permuting function add_p that was presented in the previous chapter. But it has one additional parameter that is declared to be a *size-preserving function*. Any function of type

$$(k : \text{Size}) \rightarrow \text{Nat } k \rightarrow \text{Nat } k$$

can now be passed along to addWith. The second reason is that the function has two size arguments that are permuting in the recursive call. So this is really an example that plays to the strength of our system: both the Size type and the size-change principle are needed to show termination.

5.5 Examples: Sized coinductive types

The *guardedness condition* [Coq93] could be employed for checking productivity of coinductive definitions. But in the following, we will use the `Size` type to prove productivity.

5.5.1 Sized Streams

Recall the declaration of `zeroes` from section 2.4.8:

```
cofun zeroes : Stream
  zeroes = cons zero zeroes
```

An example of an unproductive stream is `unp` :

```
cofun unp : Stream
  unp = unp
```

and \searrow $(\text{head } \text{unp})^\circ$ is not defined. Now let us transfer these declarations to the sized version of `Stream`. For the productive stream `zeroes`, we turn it into

```
cofun zeroes : (i : Size) → Stream i
  zeroes (s i) = cons i zero (zeroes i)
```

This passes the termination checker, because the call matrix

$$\text{zeroes } (<) \text{ zeroes}$$

is idempotent and decreasing. With sized types, productive corecursive definitions are now structurally recursive on the size argument. This would also pass the guardedness check: the call to `zeroes` is *guarded* by `cons`.

For the unproductive stream, we have two bad choices: The first one

```
cofun unp : (i : Size) → Stream i
  unp i = unp i
```

is type-correct, but does not pass the termination-check. The second option

```
cofun unp : (i : Size) → Stream i
  unp (s i) = unp i
```

would be accepted by the termination-checker, but is not type-correct.

Now, for a productive stream, we can look at the first element of a stream with `head`, or remove the first element of a stream with `tail` :

```
fun head : Stream ∞ → Nat
  head (cons ∞ x xs) = x
```

```
fun tail : Stream ∞ → Stream ∞
  tail (cons ∞ x xs) = xs
```

We can now define the n th element of a stream:

```
fun nth : Nat → Stream ∞ → Nat
  nth zero xs = head xs
  nth (succ n) xs = nth n (tail xs)
```

5.5.2 Fibonacci stream

The obligatory example is to define the stream of Fibonacci numbers:

```
cofun fib' : Nat → Nat → (i : Size) → Stream i
  fib' x y (s i) = cons i x (fib' y (add x y) i)
let fib : Stream ∞ = fib' (succ zero) (succ zero) ∞
```

and get the fourth Fibonacci number by:

```
let fib4 : Nat = nth (succ (succ (succ (succ zero)))) fib
```

5.5.3 Equality of streams

The following is a type-correct declaration:

```
let isEq : Eq (Stream ∞) (zeroes ∞) (cons ∞ zero (zeroes ∞))
  = refl (Stream ∞) (zeroes ∞)
```

The inferred type of `refl (Stream ∞) (zeroes ∞)` is

$$\text{Eq (Stream } \infty) \text{ (zeroes } \infty) \text{ (zeroes } \infty)$$

The type-checker has to unroll `(zeroes ∞)` once to see that the type is equal to the declared type of `isEq`. Now let's define another stream:

```
cofun zeroes2 : (i : Size) → Stream i
  zeroes2 (s s i) = cons (s i) zero (cons i zero (zeroes2 i))
```

We cannot prove

$$\text{Eq (Stream } \infty) \text{ (zeroes } \infty) \text{ (zeroes}_2 \infty)$$

because the type-checker can't unroll both `zeroes ∞` and `zeroes2 ∞` as this would lead to no progress. But we can define bisimilarity on streams [Coq93], as a sized coinductive predicate:

```
codata Bis : Size → Stream ∞ → Stream ∞ → Set
  bis : (i : Size) → (n : Nat) → (s1 : Stream ∞) → (s2 : Stream ∞)
    → Bis i s1 s2 → Bis (cons n s1) (cons n s2)
```

And now the following is a valid infinite proof:

```

cofun isBis : (i : Size) → Bis i (zeroes ∞) (zeroes2 ∞)
  isBis (s s i) =
    bis (s i) zero (cons ∞ zero (zeroes ∞)) (cons ∞ zero (zeroes2 ∞))
      (bis i zero (zeroes ∞) (zeroes2 ∞) (isBis i))

```

Observational equality [AMS07], which is a recent attempt to strengthen decidable equality for dependent types, would automatically entail this notion of bisimilarity for coinductive types.

5.5.4 Stream processors

A stream processor [GHP06] transforms an input stream into an output stream. It can `get` an element from the input stream or `put` an element into the output stream. The generated output stream is productive if the stream processor does not stop putting elements into the output stream.

Productive stream processors can be nicely modeled with mixed inductive/coinductive sized types [Abe07]. As `Mugda` does not support such mixed declarations directly, we need to resort to a *continuation-passing style*:

```

data ISP (+ K : Set) : Set
  put : Nat → K → ISP K
  get : (Nat → ISP K) → ISP K

```

```

sized codata SP : Size → Set
  isp : (i : Size) → ISP (SP i) → SP (s i)

```

As an example, the stream processor `adder` continuously gets a natural number n , then puts the sum of the following n input elements into the output stream:

```

fun iadder : Nat → Nat → (K : Set) → K → ISP K
  iadder zero acc K k = put K acc k
  iadder (succ n) acc K k = get K (λ m. (iadder n (add m acc) K k))
cofun coadder : (i : Size) → SP i
  coadder (s i) = isp i (get (SP i) (λ n. iadder n zero (SP i) (coadder i)))
let adder : SP ∞ = coadder ∞

```

The execution of a stream processor, called *eating*, follows:

```

fun ieat : (K : Set) → (C : Set) →
  ISP K → Stream ∞ → (Nat → K → Stream ∞ → C) → C
  ieat K C (get K f) (cons ∞ a as) h = ieat K C (fa) as h
  ieat K C (put K b k) as h = h b k as
cofun eat : (i : Size) → SP ∞ → Stream ∞ → Stream i
  eat (s i) (isp ∞ ip) as = ieat (SP ∞) (Stream (s i))
    ip as (λ b. λ k. λ as'. (cons i b (eat i k as')))

```

`ieat` is shown terminating due to Axiom 2, and `eat` is structurally recursive on the size argument, thus productive.

But `eat` would not be accepted by the guardedness check: Although the recursive call to `eat` is guarded by the constructor `cons`, this constructor is again surrounded by `ieat`, which is not allowed by this syntactic criterion.

5.6 Admissible recursive function declarations

As we will see, the use of `Size` needs to be constrained. Otherwise, there are declarations that are type-correct and pass the termination-checker, but lead to non-termination.

We will give a criterion for when a mutual declaration δ is *admissible*. Admissibility is a necessary concept in works on sized types. Admissibility based on *monotonicity* is for example used in [BGP06] and [Bla04]. The admissibility of [Abe06] and [HPS96] is based on the more advanced concept of *continuity*.

The criterion for *Mugda* is based on monotonicity and has to deal with dependent pattern matching. We look at the types of a mutual declaration, and also – because matching on constructors has an influence on the type – at the patterns. Consider for example the following:

```
fun bad1 : (i : Size) → Bool
  bad1 (s i) = bad1 i
```

It is type-correct, and the termination-checker will happily tell you that `bad1` is terminating. But $\searrow (\text{bad}_1 \infty)^p$ is not defined.

It can be argued that the pattern `s i` does not cover all cases: A hypothetical bottom size element would not match against `s i`. Case distinction on a size should not be accepted. For the same reason, the following function definition is also not admissible:

```
fun bad2 : (i : Size) → Nat i → Bool
  bad2 s s i (zero (s i)) = bad2 (s i) (zero i)
  bad2 s i (succ i x) = bad2 i x;
```

For example, $\searrow (\text{bad}_2 \infty (\text{zero } \infty))^p$ is not defined. Such examples are forbidden by the *bottom-check* [HPS96] in other systems with sized types.

Both examples will be rejected because of their incomplete size pattern. But there are examples where the size patterns are complete, but the type needs to be rejected. The program in figure 5.8 is adopted from [Abe06]. The type of the function `loop` is

$$(i : \text{Size}) \rightarrow \text{Nat } i \rightarrow (\text{Nat } \infty \rightarrow \text{Maybe } (\text{Nat } i)) \rightarrow \text{Bool}$$

and needs to be rejected by an admissibility check. Otherwise `diverge` could be constructed, which leads to non-termination when evaluated.

```

data Maybe (+ A : Set) : Set
  nothing : Maybe A
  just : A → Maybe A

fun shift_case : (i : Size) → Maybe(Nat(s i)) → Maybe(Nat i)
  shift_case i (nothing Nat s i) = nothing (Nat i)
  shift_case i (just Nat s i (zero i)) = nothing (Nat i)
  shift_case i (just Nat s i (succ i x)) = just (Nat i) x

let shift : (i : Size) → (Nat ∞ → Maybe (Nat (s i)))
  → Nat ∞ → Maybe (Nat i)
  = λ i. λ f. λ n. shift_case i (f (succ ∞ n))

let inc : Nat ∞ → Maybe Nat ∞ = λ n. just Nat ∞ (succ ∞ n)

mutual
  fun loop : (i : Size) → Nat i
    → (Nat ∞ → Maybe (Nat i)) → Bool
    loop s i (zero i) f = loop_case (s i) f (f (zero i))
    loop s i (succ i n) f = loop i n (shift i f)

  fun loop_case : (i : Size) → (Nat ∞ → Maybe (Nat i))
    → Maybe (Nat i) → Bool
    loop_case i f (nothing Nat i) = tt
    loop_case s i f (just Nat (s i) (zero i)) = tt
    loop_case s i f (just Nat (s i) (succ i y)) = loop i y (shift i f)

let diverge : Bool = loop ∞ (zero ∞) inc

```

Figure 5.8: Loop example

What follows is admissibility for the case of a mutual recursive declaration. Admissibility for corecursive declarations will be covered in the next section.

5.6.1 Admissible type

First, all types of a mutual function declaration have to be *admissible*. This can be described informally for a type *expression* t :

- For an inductive function with type

$$t = (a_1 : A_1) \rightarrow \dots \rightarrow (a_n : A_n) \rightarrow R$$

it is required that for every *argument type* $(a_j : A_j)$ of the form $(i : \text{Size})$

- For $k > j$, either i is not occurring in A_k or A_k is *inductive in* i , i.e. A_k is a sized inductive type of size i .
- the *result type* R is *monotone* in i .

The formal judgment in figure 5.9 is defined on evaluated types, not expressions.

$$\frac{k; \mathbf{N} \vdash v \leq [\mathbf{s} \ i/i]\{v\}}{k \vdash v \ \mathbf{mon} \ i} \quad \frac{\text{D sized inductive data type}}{k \vdash \text{D } p_1 \dots p_n \ i \ v_1 \dots v_m \ \mathbf{ind} \ i} \ \Sigma \text{D} = (v_D, n)$$

$$\frac{\frac{k \vdash v_A \ \mathbf{ind} \ i \ \text{or} \ k \vdash i \ \mathbf{nocc} \ v_A}{k+1 \vdash \searrow B^{\rho, x=k} \ \mathbf{admIndSize} \ i} \quad \frac{k \vdash v \ \mathbf{mon} \ i}{k \vdash v \ \mathbf{admIndSize} \ i}}{k \vdash \text{Pi } x \ v_A \ B^\rho \ \mathbf{admIndSize} \ i} \quad \frac{k+1 \vdash \searrow B^{\rho, x=k} \ \mathbf{admIndSize} \ k \ \text{if } v_A = \text{Size}}{k+1 \vdash \searrow B^{\rho, x=k} \ \mathbf{admIndType}}}{k \vdash \text{Pi } x \ v_A \ B^\rho \ \mathbf{admIndType}}$$

$$\frac{}{k \vdash v \ \mathbf{admIndType}} \ v \neq \text{Pi } x \ v_A \ B^\rho$$

Figure 5.9: Admissible type for inductive function

5.6.2 Size pattern coverage

As was said before, the check for complete coverage of pattern matching will not be detailed in this work. But a special case is coverage for the built-in Size type: without this check even non-terminating function declarations would be accepted. For a mutual recursive declaration δ , the patterns \vec{p} are

size complete, if no pattern of the form (sp) occurs. So only a variable i will be a complete pattern. This still allows s to appear inside of inaccessible patterns. Although straightforward, this is formalized with the judgment in figure 5.10.

$$\begin{array}{c}
 \frac{\vdash \vec{p} \text{ sizePats}}{\vdash c \vec{p} \text{ sizePat}} \quad \frac{}{\vdash x \text{ sizePat}} \quad \frac{}{\vdash \underline{e} \text{ sizePat}} \\
 \\
 \frac{\vdash p \text{ sizePat} \quad \vdash \vec{p} \text{ sizePats}}{\vdash p \vec{p} \text{ sizePats}} \quad \frac{}{\vdash \diamond \text{ sizePats}}
 \end{array}$$

Figure 5.10: Size pattern completeness for recursive function

5.6.3 Admissibility criterion

Putting the previous two sections together, the final admissibility criterion follows:

For the mutual recursive function declaration $\delta =$

mutual

$\text{fun } f_1 : A_1$
 $\quad \vec{\gamma}_1$
 \dots
 $\text{fun } f_n : A_n$
 $\quad \vec{\gamma}_n$

if for every $i \in \{1 \dots n\}$:

1. $1 \vdash \searrow A_i^\diamond \text{ admIndType}$
2. $\vdash \vec{p} \text{ sizePats}$ for every clause $f \vec{p} e \in \gamma_i$

then δ is *admissible*.

Examples

For the rejected example `loop`, i is occurring in the argument type

Maybe (Nat (s i))

but this argument type is not inductive in i , so the function `loop` is not admissible because of its type. Both `bad1` and `bad2` are not admissible because their size patterns. As can be seen, all examples given in sections 5.4 and 5.5 are admissible.

5.7 Admissible corecursive declarations

For corecursive functions, the type admissibility criterion is even more restricted. Again, it is first outlined for a type expression:

- For a corecursive function with type

$$t = (a_1 : A_1) \rightarrow \dots \rightarrow (a_n : A_n) \rightarrow R$$

it is required that for every argument type $(a_j : A_j)$ of the form $(i : \text{Size})$

- i is not occurring in argument types A_k (where $k > j$).
- the result type R is *coinductive in i* , i.e. R is a sized coinductive type of size i .

This is formalized with the judgment given in figure 5.11.

We do not check the completeness of the size pattern for corecursive declarations because the size argument is determined by the right hand side – the object that is being defined – and thus can be seen as inaccessible.

$$\frac{\text{D sized codata}}{k \vdash \text{D } p_1 \dots p_n \ i \ v_1 \dots v_m \ \mathbf{coind} \ i} \Sigma \text{D} = (v_D, n)$$

$$\frac{k \vdash i \ \mathbf{noCC} \ v_A \quad k+1 \vdash \searrow B^{\rho, x=k} \ \mathbf{admCoSize} \ i}{k \vdash \text{Pi } x \ v_A \ B^\rho \ \mathbf{admCoSize} \ i} \quad \frac{k \vdash v \ \mathbf{coind} \ i}{k \vdash v \ \mathbf{admCoSize} \ i}$$

$$\frac{k+1 \vdash \searrow B^{\rho, x=k} \ \mathbf{admCoSize} \ k \ \text{if } v_A = \text{Size} \quad k+1 \vdash \searrow B^{\rho, x=k} \ \mathbf{admCoType}}{k \vdash \text{Pi } x \ v_A \ B^\rho \ \mathbf{admCoType}} \quad \frac{}{k \vdash v \ \mathbf{admCoType}}$$

Figure 5.11: Admissibility type for corecursive function

5.7.1 Admissibility criterion

For the mutual corecursive function declaration $\delta =$

mutual
 $\text{cofun } f_1 : A_1$
 $\vec{\gamma}_1$
 \dots
 $\text{cofun } f_n : A_n$
 $\vec{\gamma}_n$

if for every $i \in \{1 \dots n\}$:

- $1 \vdash \searrow A_i^\diamond$ **admCoType**

then δ is *admissible*.

5.7.2 Fibonacci à la Haskell

Both admissibility criteria do not allow an argument of the form `Stream i`. Functions can only have arguments of the form `Stream ∞`, so pattern matching can only happen on fully constructed infinite objects.

In the lazy functional programming language Haskell [Je99], which supports partiality, the following is a valid definition of the stream of Fibonacci numbers:

```
fib :: [Int]
fib = (1 : ( 1 : zipWith (+) fib (tail fib)))
```

Translating this definition into Mugda as a sized stream is not possible, and indeed is not productive with the given evaluation semantics.

In Haskell, the helper functions `zipWith` and `tail` can operate on any list, even on `fib` which is just being defined. In the Mugda setting, `tail` would need to have the type

$$(i : \text{Size}) \rightarrow \text{Stream } (s \ i) \rightarrow \text{Stream } i$$

This type is not admissible, because i is occurring in the argument type `Stream (s i)`.

5.8 On the necessity of subtyping

Subtyping for inductive types is actually just for convenience to the user. For example

```
fun weakSNat : (i : Size) → Nat i → Nat (s i)
  weakSNat si (zero i) = zero (s i)
  weakSNat si (succ i x) = succ (s i) (weakSNat i x)
```

could be used to manually weaken an object into a greater type. But subtyping does make the system more comfortable, and also this weakening would have a significant impact on runtime performance. Its bad brother

```
fun badSNat : (i : Size) → Nat (s i) → Nat i
  badSNat si (zero (s i)) = zero i
  badSNat si (succ (s i) x) = succ i (badSNat i x)
```

fails to pass the admissibility test. For `Stream`, the weakening function would be

cofun $\text{weakStream} : (A : \text{Set}) \rightarrow (i : \text{Size}) \rightarrow \text{Stream } A \ (s \ i) \rightarrow \text{Stream } A \ i$
 $\text{weakStream } \underline{A} \ \underline{s \ i} \ (\text{cons } A \ ((s \ i) \ x \ xs) = \text{cons } A \ i \ x \ (\text{weakStream } A \ i \ xs))$

but its type is not admissible. This seems to indicate that a less restrictive admissibility criterion should be achievable.

5.9 Putting it all together

Now that we have limited the use of the `Size` type, we can formulate our termination criterion for `Mugda` with sized types. We conjecture:

Proposition : Size-change principle for `Mugda` with sized types

Given

1. Σ a terminating signature
2. a mutual (co)recursive declaration δ
3. Σ' is resulting from type-checking δ in Σ
4. δ is admissible
5. every idempotent call-matrix $\alpha \in \text{complete cs}(\delta)$ is decreasing

then Σ' is terminating.

The integration of the size type in `Mugda` went through a lot of changes. At first, `Size` was a small type. Through this, a “bad user” could do a lot of things to fool the system. Taking `Size` out of `Set` defused the situation quite a bit. Admissibility was at times much more lenient, until counter-examples such as “Fibonacci à la Haskell” were found.

Chapter 6

Conclusion

We think that the *Mugda* language demonstrates the usefulness of sized types. To that end, we have stretched Coquand's simple type-checking algorithm to a quite usable system. To our knowledge, *Mugda* is the first system to combine the sized type approach with the size-change principle.

Important for *Mugda* as a proof system would be to add coverage checking of patterns, and to proof all those propositions that this thesis only tried to make look somewhat plausible. As inductive families seem to add quite a bit complexity, this is likely not easy.

Anyway, we hope this work also provides an easy access to both dependent types and the sized type approach. The pattern matching notation used by *Mugda* should be a little bit more accessible than the fixed point operators used in a lot of previous presentations of sized types.

Finally, here are some further ideas that came up during development of this thesis:

Inference of sizes The possibility to infer all size annotations automatically should be explored. The system described in [BGP06] is able to infer size annotations with only minimal assistance required by the user. It remains to be seen if and how this inference could be applied to *Mugda*.

Implicit arguments At runtime, the size arguments all end up being ∞ . So it would be wasteful if it was kept around during execution of a program. The removal of type information that is not needed at runtime is ongoing research [Miq01, BMM03]. As lists, vectors and sized list behave the same, conversion functions can be written by the user to change from one representation to the other. At runtime, these could be safely removed as they would amount to identity functions.

More mutual definitions We could allow more combinations for mutual declarations. Defining an inductive data type together with a recursive

function amounts to the so-called inductive-recursive definitions of [DS01]. But also mixed inductive/coinductive definitions should be interesting. This direction was investigated and some adjustment of the admissibility of *Mugda* seems to be necessary to handle such definitions.

Better admissibility The current admissibility criterion can probably be relaxed somewhat to allow more definitions. More audacious would be to adapt the more advanced concepts from [Abe06].

Higher-order subtyping Along with better admissibility, higher-order subtyping would enable the definition of functions that are parameterized over any sized type. Polarized subtyping [Ste98] could be explored, which is already used in the sized polymorphic lambda calculus of [Abe06].

Future of sized types We think it could be worthwhile to integrate a sized type approach into a full system like *Agda2* [Nor07], especially with an extension to coinductive types in mind. Productivity is handled quite naturally with a sized type approach. Furthermore, there are plans to give a translation of the full *Agda2* language to a simpler core language [CKNT07], which can be more easily justified. Once again, a sized type approach could be considered for this core language.

Appendix A

Mugda implementation

Mugda was implemented in the function language Haskell [Je99], using the Glasgow Haskell compiler (GHC) [JHH⁺93].

A.1 Source file listing

- `Lexer.x` : the alex lexer file
- `Parser.y` : the happy parser file
- `Concrete.hs` : concrete syntax produced by the parser
- `TraceError.hs` : provides the user a trace when an error occurs
- `ScopeChecker.hs` : turns concrete into abstract syntax
- `Abstract.hs` : produced by the scope-checker, used during type-checking
- `Values.hs` : defines `values`, `evaluation`, `signature`, `type-check monad`
- `TypeChecker.hs` : type-checking with admissibility
- `Termination.hs` : syntactic termination check
- `Completeness.hs` : size pattern completeness check
- `SPos.hs` : strict positivity checker
- `Main.hs` : the main module
- `example` directory: example input files
- `Makefile` : for compilation

A.2 Usage

Mugda as presented was pretty much directly transferred to ASCII syntax:

- lists of constructors and clauses are grouped with brackets `{ }` and separated with semicolon `;`
- $(x : A) \rightarrow B$ is written `(x : A) -> B`
- $A \rightarrow B$ is written `A -> B`
- $\text{let } x : A = e \text{ in } f$ is written `let x : A = e in f`
- $\lambda x. e$ is written `\x -> e`
- \underline{e} is written `.e`
- ∞ is written `#`
- \mathbb{S} is written `$`
- one line comments are prefixed by `--`
- multi-line comments are put between `{-` and `-}`

a `let` declaration can be prefixed with `eval`. Then the value will be evaluated after type checking is done. As an example showing most of the syntactical features, here is the Fibonacci stream example (`examples/fib.ma`) in text format:

```
data Nat : Set {
  zero : Nat;
  succ : Nat -> Nat
}

fun add : Nat -> Nat -> Nat {
  add zero = \y -> y;
  add (succ x) = \y -> succ (add x y)
}

sized codata Stream : Size -> Set {
  cons : (i : Size) -> Nat -> Stream i -> Stream ($ i)
}

fun tail : Stream # -> Stream # {
  tail (cons .# x xs) = xs
}
```

```

fun head : Stream # -> Nat {
  head (cons .# x xs) = x
}

fun nth : Nat -> Stream # -> Nat {
  nth zero xs = head xs;
  nth (succ x) xs = nth x (tail xs)
}

let 1 : Nat = (succ zero)

cofun fib' : (x : Nat ) -> (y : Nat )
           -> (i : Size ) -> Stream i {
  fib' x y ($ i) = cons i x (fib' y (add x y) i)
}

-- fib = 1, 1, 2, 3, 5 , 8 ...
let fib : Stream # = (fib' 1 1 #)

let 4 : Nat = (succ (succ (succ 1)))

-- fib(4) = 5
eval let fib4 : Nat = nth 4 fib

```

Running `Main examples/fib.ma` yields the console output:

```

***** Mugda v1.0 *****
--- scope checking ---
--- type checking ---
--- evaluating ---
fib4 evaluates to (succ (succ (succ (succ (succ zero))))))

```

A.3 Some implementation details

The `alex` [Mar07a] and `happy` [Mar07b] tools were used to generate lexer and parser for Mugda. Most of the Haskell code is monadic, where *monad transformers* [Gra06] are used to keep the signature in a state monad, to provide I/O and tracing of errors. The execution of Mugda can be broken into 4 stages:

Parsing The input file is parsed into *concrete syntax*.

Scope-Checking As mentioned in chapter 3, scope-checking is the first step after parsing. During parsing, it is not known whether an identifier is

a variable or a constructor etc. If the Mugda program is well-scoped, scope-checking produces *abstract syntax* where all identifiers are categorized. In addition, some syntactic tests like checking linearity of patterns are done during this stage.

Type-Checking Every declaration is type-checked. For mutual declarations, the type-checker also checks admissibility and finally invokes the termination-checker.

Evaluation For all declarations of the form `eval let l : A = e`, `e` is evaluated and this value is displayed.

Bibliography

- [AA02] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.
- [Abe04] Andreas Abel. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS’03).
- [Abe06] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.
- [Abe07] Andreas Abel. Mixed inductive/coinductive types and strong normalization, to appear. In *The Fifth ASIAN Symposium on Programming Languages and Systems (APLAS 2007)*, 2007.
- [AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *PLPV*, pages 57–68. ACM, 2007.
- [Aug85] Lennart Augustsson. Compiling pattern matching. In *FPCA*, pages 368–381, 1985.
- [Aug98] Lennart Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.
- [BFG⁺04] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14:97–141, February 2004.
- [BGP06] G. Barthe, B. Grégoire, and F. Pastawski. Type-based termination of recursive definitions in the calculus of inductive constructions. In *Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR’06)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, November 2006. To appear.

- [Bla04] F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems, 2004.
- [BMM03] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *TYPES*, pages 115–129, 2003.
- [CKNT07] Thierry Coquand, Yoshiki Kinoshita, Bengt Nordström, and Makoto Takeyama. A simple type theoretic language: Mini-TT (unpublished). 2007.
- [Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.
- [Coq93] Thierry Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs (TYPES '93)*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.
- [Coq96] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, 1996.
- [DS01] Peter Dybjer and Anton Setzer. Indexed induction-recursion. *Lecture Notes in Computer Science*, 2183:93–??, 2001.
- [Dyb94] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.
- [GHP06] Neil Ghani, Peter Hancock, and Dirk Pattinson. Continuous functions on final coalgebras. *Electr. Notes Theor. Comput. Sci.*, 164(1):141–155, 2006.
- [Gim98] E. Gimenez. A tutorial on recursive types in coq, 1998.
- [GMM06] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday (Goguen Festschrift)*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.
- [Gra06] Martin Grabmüller. Monad Transformers Step by Step. Draft paper, October 2006.
- [HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Symposium on Principles of Programming Languages*, pages 410–423, 1996.

- [HS95] M. Hofmann and T. Streicher. The groupoid interpretation of type theory, 1995.
- [INR07] INRIA. The coq proof assistant. <http://coq.inria.fr>, 2007.
- [Je99] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language. Technical report, February 1999.
- [JHH⁺93] S. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The glasgow haskell compiler: a technical overview, 1993.
- [Lan63] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.
- [LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36(3):81–92, 2001.
- [MAG07] Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In *The Australasian Theory Symposium (CATS2007)*, January 2007.
- [Mar07a] Simon Marlow. Alex – a lexical analyser generator for haskell. <http://haskell.org/alex>, 2007.
- [Mar07b] Simon Marlow. Happy – the parser generator for haskell. <http://haskell.org/happy>, 2007.
- [McB07] Conor McBride. Epigram, <http://e-pig.org>, 2007.
- [Miq01] Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In S. Abramsky, editor, *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'01, Krakow, Poland, 2–5 May 2001*, volume 2044, pages 344–359. Springer-Verlag, Berlin, 2001.
- [ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, Napoli, 1984.
- [Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
- [NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *Int. Series of Monographs on Computer Science*. Oxford, 1990.

- [PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. F. Groote, editors, *Proceedings 1st Int. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993*, volume 664, pages 328–345. Springer-Verlag, Berlin, 1993.
- [Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, 1994.
- [PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.
- [SP03] C. Schurmann and F. Pfenning. A coverage checking algorithm for lf, 2003.
- [Ste98] Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Technische Fakultät, Universität Erlangen, 1998.
- [Wah07] David Wahlstedt. *Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion*. PhD thesis, Chalmers University of Technology, 2007. ISBN 978-91-7291-979-2.
- [XP99] Howgwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.