Ludwig-Maximilians-Universität München
Institut für Informatik

Type Checking Without Types

Matthias Benkard

April 16, 2013

Zusammenfassung

Wir stellen TypeCore vor, eine funktionale Sprache, deren Typsystem grob an abhängige Typen angelehnt ist, jedoch auf Mereologie anstelle von Mengenlehre basiert. Diese Neuerung macht die Trennung zwischen Typen und *kinds* sowie das Vorhandensein eines Set-Typen überflüssig. In unserem Formalismus ist Typisierung zu Subtyping äquivalent: Ein Wert ist genau dann ein Bewohner des Typen α, wenn er ein Subtyp von α ist.

Da jeder Wert zugleich ein Typ ist, ist es unnötig, syntaktisch zwischen Typen und Programmcode zu unterscheiden. Insbesondere lassen sich algebraische Datentypen durch denselben Fixpunktmechanismus definieren wie rekursiv definierte Funktionen. Ebenso sind Pfeiltypen Funktionen, die wie ihre „Bewohner“ Musterangleich für Fallunterscheidungen einsetzen. Σ-Typen schließlich sind durch Komprehensionen darstellbar.

Abstractum

Proponimus linguam ordinatri TypeCore, systema classificationis eius, a classificationes dependentes quidem derivata, non copiis constructa, sed qualitate mereologica est. In lingua ea, omnis res certe classificatione, si subclassificatio illius est.

Omnibus rebus classificationibus, syntax speciala classificationium non necesse est. Exemplo gratia, classificationes datorum algebraicorum puncto stabile definire potes, quomodo functiones recursivas definis. Simile, classificationes existentiales comprehensionibus, universales functionibus expressibiles sunt.

Abstract

We introduce TypeCore, a programming language with a type system roughly based on dependent types, but built on top of mereology rather than set theory (and thus without the need for a Set type). In our formalism, typing is equivalent to subtyping; that is, a value is a member of the type α iff it is a subtype of α.

Since every value is a type, there is no need for special syntax for the declaration or definition of types. In particular, algebraic data types can be defined through the same fixed-point mechanism that is used for recursive function definitions. Similarly, arrow types are functions, employing pattern matching just as their “members” do. Finally, Σ-types can be represented by object comprehensions.

Acknowledgments

I would like to thank my advisor, Andreas Abel, for his help and patience. He spent an extraordinary amount of time dispelling my regular confusion and providing me with valuable input.

Helmut Schwichtenberg introduced me to mathematical logic and frequently encouraged me to investigate more advanced topics.

The department graciously funded my first encounter with international scientists, which was an inspiring and important experience. I would like especially to thank Martin Hofmann for introducing me to the program and Hans JÃijrgen Ohlbach for his efforts in providing me with the necessary information and motivation.

Finally, I thank my supervisor, Martin Hofmann, who gave me encouragement not just in writing this diploma thesis, and who has acquired some of the most helpful and inspiring research fellows in his position of chair head.

Contents

Listings

1 Introduction

In dependent type theory, originally developed by Martin-Löf [PM84], programmers can do type-level calculation and have types depend on values. There is, however, traditionally still a clear distinction between (compile-time, static) types and (run-time, dynamic) values. In AGDA [BDN09], for example, types cannot be the result of a run-time expression. Neither can the input of a program be a type.

This strict separation between types and values is rooted in set theory. In ZFC [Jec11], for example, the foundation axiom mandates that some kind of type hierarchy needs to exist, with atomic values at the bottom and an arbitrarily large tower of sets on top.

But set theory, while an extremely useful foundation of mathematics, is quite limited as applied to the physical world. Sets are, after all, an abstract concept with no physical counterpart. For is anything really a *set* of things, rather than an arbitrary collection without any kind of hierarchy in between the components? One would be hard-pressed to suggest, for instance, that the room I am located in while I am writing this text has the chair I am sitting on as an *element*, whose *elements*, again, include its legs, which are somehow not elements of the room directly. Indeed, parthood is naturally transitive, and it would be quite elegant to be able to say that every object is part of itself, which cannot be modeled adequately using the ϵ-relation of set theory. Ancient philosophical wisdom notwithstanding, in the physical world, the whole really is the sum of its parts!

Because of this impedance mismatch, axiomatic mereology [CV99] has been proposed as an alternative to set theory, particularly in fields related to topology.

We suggest that mereology might not only be a good fit for physical and geometrical modelling, but also for modelling computation. For one, database systems have long relied on the relational algebra and relational calculus for their models. Both formalisms are usually described as set-theoretical, but in fact, instead of elements, operate purely on non-nested collections. Indeed, the join operator of the relational algebra, arguably its most significant contribution above the use of set theory itself, is precisely a non-nested cartesian product. All things considered, one cannot help but wonder whether a mereological underpinnings would be more appropriate than ZFC for relational models.

In addition, non-deterministic computation, as implemented in logic-oriented languages like PROLOG [SS94], handles collections of both input and output values in much the same way as the relational calculus, namely by way of joins, unions and intersections. Here, too, mereology could prove to be an appropriate semantical foundation.

In this work, we explore a slightly different angle, namely static typing of purely functional, mereological programs. To this effect, we propose a variation on the λ-calculus with a mereological semantics.

Our type system is loosely based on dependent types, but tailored towards the mereological case. Instead of the ϵ-relation, we base our type judgments on parthood, i. e., a value v is of type A if and only if it is a part of type A. One immediate consequence is the absence of any distinction between types and their inhabitants: Every value is a type containing itself and all of its parts; and conversely, every type is a composite value consisting of its inhabitants.

2 Syntax

2.1 Grammar

The language grammar is deliberately minimal. There are two kinds of literals *lit*: Apart from integers, we have LISP-like symbols of the form $'ident$ that serve as a replacement for type and data contructors. In addition to the pair constructor, there is a second infix operator $\oplus$ that is used to denote a mereological fusion. Finally, functions are composed of *(pattern, expression)* pairs where the pattern can include bounds ($p <: \tau$) that restrict argument types.

$$
\begin{array}{llllll}
\mathsf{Exp} & \ni & e, f, A, B & ::= & x & \text{Bound variable} \\
 & & & \mid & a & \text{Parameter} \\
 & & & \mid & c & \text{Literal} \\
 & & & \mid & e, e' & \text{Pair} \\
 & & & \mid & f\, e & \text{Application} \\
 & & & \mid & \mathsf{fun}\ \{\overrightarrow{p \to e}\} & \text{Function} \\
 & & & \mid & \mathsf{let}\ \sigma\ \mathsf{in}\ e & \text{Local bindings} \\
 & & & \mid & \mathsf{fix}\ p_x \to e & \text{Fixed-point (recursion)} \\
 & & & \mid & \bigoplus \vec{e} & \text{Fusion} \\
 & & & \mid & e <: A & \text{Restriction (type annotation)} \\
 & & & \mid & \{p\} & \text{Comprehension} \\
 & & & & & \\
\mathsf{Lit} & \ni & c & ::= & 'alpha \mid 0 \mid 1 \mid 2 \mid \ldots & \\
\mathsf{Pat} & \ni & p, q & ::= & x \mid c \mid p, q \mid p <: A & \text{Pattern} \\
\mathsf{VarPat} & \ni & p_x & ::= & x \mid x <: A & \text{Variable pattern} \\
\mathsf{Env} & \ni & \sigma & ::= & \cdot \mid \sigma, x = e & \text{Substitution}
\end{array}
$$

We further define a number of abbreviations.

$$
\begin{array}{lll}
\top & := & \{x\} \\
\bot & := & \bigoplus\{\} \quad \text{(i. e., the empty fusion)} \\
\mathsf{fun}\ \{\vec{p} \to e\} & := & \mathsf{fun}\ \{p_1 \to \mathsf{fun}\ \{p_2 \to \cdots \mathsf{fun}\ \{p_n \to e\} \cdots\}\} \\
\mathsf{let}\ x\ \vec{p} = e & := & \mathsf{let}\ x = \mathsf{fun}\ \{\vec{p} \to e\}
\end{array}
$$

2.2 Examples

2.2.1 Natural numbers

To illustrate the way the language works, consider Listing 2.1 in the dependently typed language AGDA.

Listing 2.1 Peano-natural numbers in Agda.

```
data Nat : Set where
  Zero : Nat
  Succ : Nat → Nat
```

Here, we define an algebraic data type called *Nat* consisting of two constructors *Zero* and *Succ*, the latter taking another value of type *Nat* as an argument. We use this type to model the natural numbers in the usual Peano way.

In addition, we define a data type *Vec*, modelling a space of lists, which encodes the length of its elements, the type of which is given as an argument to the type constructor, using an argument of type *Nat*. Thus, *Vec* is a dependent type, relying on a constructor *Cons* doing type-level computation.

In TypeCore, we can approximate the definition of *Nat* by the use of pair construction, disjunction, and recursion (Listing 2.2).

Listing 2.2 Peano-natural numbers in TypeCore.

```
let nat = fix nat → 'zero ⊕ ('s, nat)
```

(In the following, we will be omitting the explicit fixed-point for implied recursion in all example definitions.)

This expression evaluates to $'zero \oplus ('s, ('zero \oplus ('s, 'zero \oplus ...)))$, which is equivalent to $'zero \oplus ('s, 'zero) \oplus ('s, ('s, 'zero)) \oplus ...$, as close to the set of naturals as we can get using mereological sums. We now have $'zero <: nat$, $('s, 'zero) <: nat$, and so on, as in the set-theoretical case. In addition, we also have $'zero \oplus ('s, 'zero) <: nat$, et cetera.

2.2.2 Vectors

One particularly traditional example of dependent typing is the type of vectors of a specified length n containing elements of a given type a. An implementation in Agda is given in Listing 2.3.

Listing 2.3 A type of fixed-length vectors in Agda.

```
data Vec (a : Set) : Nat → Set where
  Nil  : Vec a Zero
  Cons : (n : Nat) → a → Vec a (Succ n)
```

A TypeCore translation is possible by defining a recursive function taking the element type and length as arguments (Listing 2.4).

Listing 2.4 A type of fixed-length vectors in TypeCore.

```
let vec = fun {a → fun {'zero → 'nil; ('s,n) → (a, vec a n)}}
```

Note that in the TypeCore version, we have omitted specifying the second argument to be of type *nat*, which we could have done by annotating both clauses of the inner function by a pattern bound.

2.2.3 Destructuring

Vectors and natural numbers as defined above can be processed by pattern matching, as usual (Listing 2.5).

Listing 2.5 A type-safe `first` function in TypeCore.

```
let first = fun {a n (x, xs) → x}
  <: fun {a n (v <: vec a ('s, n)) → a}
```

If our system is to succeed in posing an alternative to dependent types, it must be able to determine that *first* is indeed a function that maps non-empty vectors to their element types.

2.2.4 Comprehensions and Σ-types

As a final example of the basic syntactic correspondence between TypeCore and set-based, dependently typed programming languages, consider the Agda code fragment given in Listing 2.6 defining an abstract type comprised of pairs of naturals n and vectors of the corresponding length.

Listing 2.6 A type of vectors paired with their lengths in Agda.

```
data ArbVec (a : Set) : Set where
  arbVec : (n : Nat) → Vec a n → ArbVec a
```

Since there are no dependent pairs in TypeCore, we use an object comprehension instead. The corresponding code is given in Listing 2.7.

Listing 2.7 A type of vectors paired with their lengths in TypeCore. $\texttt{arbvec}\ a = \{n <: \texttt{nat},\ v <: \texttt{vec}\ n\ a\}$ is taken to be the fusion $\bigoplus_{n \le nat, v \le (vec\ n\ a)} (n, v)$.

```
let arbvec = fun {a → {(n <: nat, v <: vec n a)}}
```

Now, note that $('s, nat)$ is the type of natural numbers ≥ 1. From this, it is easy to see how to define a subtype of *arbvec* containing only non-empty vectors. See Listing 2.8 for the implementation.

Listing 2.8 A type of non-empty vectors paired with theirs lengths in TypeCore

```
let arbvec1 = fun {a → {(n <: ('s, nat), v <: vec n a)}}
```

In Listing 2.9, we also rewrite $first$ to take objects of type $arbvec1$ as arguments.

Listing 2.9 Another safe definition of `first` in TypeCore.

```
let first' = fun {a (('s, n), (x, xs)) → x}
```

Given these definitions, we expect our system to be able to prove that $first' <:$ fun $\{a\ (v <: arbvec1\ a) \to a\}$.

3 Evaluation

3.1 Definitions

Definition 1. Let $FV(e)$ be the set of free variables of e, $Par(e)$ the set of parameters occurring in e.

A list Γ of the form

$$\begin{array}{llll} \mathsf{Ctx} \ni \Gamma, \Delta & ::= & \cdot & \text{empty context} \\ & | & \Gamma, a <: A & \text{bound} \end{array}$$

is called a *context* iff for every initial slice $(\Gamma', a <: A) \subset \Gamma$, we have $\mathsf{FV}(A) = \emptyset$ and $\mathsf{Par}(A) \subset \mathsf{dom}(\Gamma')$.

Definition 2. A list ρ of the form

$$\begin{array}{llll} \mathsf{Subst}, \mathsf{Env} \ni \rho & ::= & \cdot & \text{empty substitution} \\ & | & \rho, x = e & \text{value binding} \end{array}$$

is called a *substitution*. Moreover, it is called an *environment* iff for every pair $(x = e) \in \rho$, $FV(e) = \emptyset$ (a notable special case being the parameter case $x = a$).

Convention 3. *A substitution used in a* let*-expression need not be an environment. Substitutions passed to judgment predicates, however, are generally assumed to be environments.*

3.2 Syntactic Classes.

We introduce several syntactic classes for various kinds of expressions:

$$\begin{array}{lll l}
E & ::= & \cdot \mid E\, e \mid w_f\, S & \text{evaluation context} \\
S & ::= & E \mid (S, e) \mid (\mathsf{let}\ \rho\ \mathsf{in}\ p, S) & \text{matching context} \\
n & ::= & E[a] & \text{neutral value} \\
\\
w & ::= & n \mid c \mid \mathsf{let}\ \rho\ \mathsf{in}\ v & \text{closure} \\
v & ::= & v_f \mid e, e' \mid \{p\} & \text{value} \\
\\
w_f & ::= & \mathsf{let}\ \rho\ \mathsf{in}\ v_f & \text{function closure} \\
v_f & ::= & \mathsf{fun}\ \{\overrightarrow{p \to e}\} \mid \mathsf{fix}\ p_x \to e & \text{function value} \\
\\
w_u & ::= & n \mid c \mid \mathsf{let}\ \rho\ \mathsf{in}\ v_u & \text{non-recursive closure} \\
v_u & ::= & \mathsf{fun}\ \{\overrightarrow{p \to e}\} \mid e, e' \mid \{p\} & \text{non-recursive value}
\end{array}$$

3.3 Evaluation.

Evaluation is given by two non-deterministic relations:

$\Gamma \vdash (\!\vert e\vert\!)_\rho \searrow w$	Expression e can evaluate to w in environment ρ, rounding wrt. Γ
$\Gamma \vdash (\!\vert e\vert\!)_\rho \searrow^! w$	Expression e can evaluate to non-fixed-point value w_u in environment ρ, rounding wrt. Γ

Free variables are interpreted according to the environment ρ. The context Γ is used to round parameters to their bounds whenever evaluation would get stuck otherwise. Parameters are, however, kept as-is when there is no need for rounding.

In case a parameter in need of rounding to proceed is not present in Γ, evaluation gets stuck, yielding a neutral value. Otherwise, the result is a value of the form v.

We introduce two abbreviations for evaluation of closures (or values):

$$\begin{array}{lcll}
\Gamma \vdash w \searrow w' & \iff & \Gamma \vdash (\!|w|\!)_{\cdot} \searrow w' & \text{Evaluation of closure } w \\
\Gamma \vdash w \searrow^! w' & \iff & \Gamma \vdash (\!|w|\!)_{\cdot} \searrow^! w' & \text{Forcing of closure } w
\end{array}$$

Evaluation is defined inductively by the following rules:

$$\frac{\Gamma \vdash \rho(x) \searrow w}{\Gamma \vdash (\!|x|\!)_\rho \searrow w} \qquad \frac{}{\Gamma \vdash (\!|a|\!)_\rho \searrow a} \qquad \frac{}{\Gamma \vdash (\!|c|\!)_\rho \searrow c} \qquad \frac{}{\Gamma \vdash (\!|v|\!)_\rho \searrow \mathsf{let}\ \rho\ \mathsf{in}\ v}$$

$$\frac{}{\Gamma \vdash (\!|(e_1, e_2)|\!)_\rho \searrow \mathsf{let}\ \rho\ \mathsf{in}\ (e_1, e_2)}$$

$$\frac{\Gamma \vdash (\!|e_i|\!)_\rho \searrow w}{\Gamma \vdash (\!|\oplus\vec{e}|\!)_\rho \searrow w} \qquad \frac{\Gamma \vdash (\!|e|\!)_{\rho, \mathsf{let}\ \rho\ \mathsf{in}\ \sigma} \searrow w}{\Gamma \vdash (\!|\mathsf{let}\ \sigma\ \mathsf{in}\ e|\!)_\rho \searrow w} \qquad \frac{\Gamma \vdash (\!|e|\!)_\rho \searrow w}{\Gamma \vdash (\!|e <: A|\!)_\rho \searrow w}$$

$$\frac{\Gamma \vdash (\!|f|\!)_\rho \searrow^! \mathsf{let}\ \rho'\ \mathsf{in\ fun}\ \{\overrightarrow{p \to e}\} \qquad \Gamma \vdash (\!|e|\!)_\rho\ !\ p_i \searrow \rho'' \qquad \Gamma \vdash (\!|e_i|\!)_{\rho', \rho''} \searrow w}{\Gamma \vdash (\!|f\ e|\!)_\rho \searrow^! w}$$

Stuck evaluation. Evaluation can get stuck whenever computation needs to be done on neutral values such as parameters:

$$\frac{\Gamma \vdash (\!|f|\!)_\rho \searrow^! n}{\Gamma \vdash (\!|f\ e|\!)_\rho \searrow n\,(\mathsf{let}\ \rho\ \mathsf{in}\ e)}$$

$$\frac{\Gamma \vdash (\!|f|\!)_\rho \searrow^! w_f \qquad w_f = \mathsf{let}\ \rho'\ \mathsf{in\ fun}\ \{\overrightarrow{p \to e}\} \qquad \Gamma \vdash (\!|e|\!)_\rho\ !\ p_i \searrow S[a]}{\Gamma \vdash (\!|f\ e|\!)_\rho \searrow w_f\ S[a]}$$

Recursion unrolling. Finally, recursion-unrolling evaluation is defined in the straight-forward way:

$$\frac{\Gamma \vdash (\!|e|\!)_\rho \searrow w_u}{\Gamma \vdash (\!|e|\!)_\rho \searrow^! w_u}$$

$$\frac{\Gamma \vdash (\!|e|\!)_\rho \searrow w_f \qquad w_f = \mathsf{let}\ \rho' \ \mathsf{in}\ \mathsf{fix}\ p_x \to e' \qquad \Gamma \vdash (\!|e'|\!)_{\rho', x = w_f} \searrow^! w_u}{\Gamma \vdash (\!|e|\!)_\rho \searrow^! w_u}$$

Example 4.

$$\begin{array}{lll} \vdash \mathsf{fix}\ nat \to {}'zero \oplus ({}'s, nat) & \searrow & \mathsf{fix}\ nat \to ({}'zero \oplus ({}'s, nat)) \\ \vdash \mathsf{fix}\ nat \to {}'zero \oplus ({}'s, nat) & \searrow^! & \mathsf{let}\ \{nat = \mathsf{fix}\ nat \to ...\}\ \mathsf{in}\ ({}'s, nat) \\ \vdash \mathsf{fix}\ nat \to {}'zero \oplus ({}'s, nat) & \searrow^! & {}'zero \end{array}$$

3.4 Pattern matching.

Pattern matching consists of attempting to destructure an expression e such that the given pattern p is matched. The result is either an environment σ binding the pattern variables of p to the corresponding values arising from the pattern match, or a stuck-match context.

$\Gamma \vdash (\!|e|\!)_\rho \ !\ p \searrow \sigma$	Pattern matching e in ρ against p succeeds with substitution σ
$\Gamma \vdash (\!|e|\!)_\rho \ !\ p \searrow S[a]$	Pattern matching e in ρ against p is stuck

Successful matches. Successful matches of an expression e against a pattern p always yield an environment σ as their result. σ can be used to bind the variables of p according to e.

$$\frac{}{\Gamma \vdash (\!|e|\!)_\rho \ !\ x \searrow (x = \mathsf{let}\ \rho\ \mathsf{in}\ e)} \qquad \frac{\Gamma \vdash (\!|e|\!)_\rho \ !\ p \searrow \sigma}{\Gamma \vdash (\!|e|\!)_\rho \ !\ (p <: A) \searrow \sigma}$$

$$\frac{\Gamma \vdash (\!|e|\!)_\rho \searrow^! c}{\Gamma \vdash (\!|e|\!)_\rho \ !\ c \searrow \cdot} \qquad \frac{\Gamma \vdash (\!|e|\!)_\rho \searrow^! \mathsf{let}\ \rho'\ \mathsf{in}\ (e_1, e_2) \qquad \Gamma \vdash (\!|e_i|\!)_{\rho'} \ !\ p_i \searrow \rho_i \text{ for } i = 1, 2}{\Gamma \vdash (\!|e|\!)_\rho \ !\ (p_1, p_2) \searrow (\rho_1, \rho_2)}$$

Stuck matches. Matches get stuck whenever a neutral value is matched against a non-trivial pattern and there is no way of rounding the value such that pattern matching can be guaranteed to either succeed or fail. In this case, the result is a matching context, which is a specific form of neutral value.

$$\frac{\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \searrow^{!} n}{\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \mathbin{!} c \searrow n} \qquad \frac{\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \searrow^{!} n}{\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \mathbin{!} (p_1, p_2) \searrow n}$$

$$\frac{\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \searrow^{!} \mathsf{let}\ \rho'\ \mathsf{in}\ (e_1, e_2) \qquad \Gamma \vdash \llparenthesis e_1 \rrparenthesis_{\rho'} \mathbin{!} p_1 \searrow S[a]}{\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \mathbin{!} (p_1, p_2) \searrow (S[a], \mathsf{let}\ \rho'\ \mathsf{in}\ e_2)}$$

$$\frac{\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \searrow^{!} \mathsf{let}\ \rho'\ \mathsf{in}\ (e_1, e_2) \qquad \Gamma \vdash \llparenthesis e_1 \rrparenthesis_{\rho'} \mathbin{!} p_1 \searrow \rho_1 \qquad \Gamma \vdash \llparenthesis e_2 \rrparenthesis_{\rho'} \mathbin{!} p_2 \searrow S[a]}{\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \mathbin{!} (p_1, p_2) \searrow \big((\mathsf{let}\ \rho_1\ \mathsf{in}\ p_1)\,, S[a]\big)}$$

Rounding match. Whenever evaluation might get stuck because of a pattern match against a neutral value, it can instead round the neutral value using the context Γ and proceed.

$$\frac{\Gamma \vdash v \searrow^{!} E[a] \qquad a \in \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \llparenthesis E[\Gamma(a)] \rrparenthesis_\sigma \mathbin{!} p \searrow \rho}{\Gamma \vdash \llparenthesis v \rrparenthesis_\sigma \mathbin{!} p \searrow \rho}$$

Note 5. *Matching cannot get stuck if* Γ *contains bounds for all parameters occurring in e. In this case, the result will always be an environment.*

Example 6.

$$\begin{array}{rcccccl} \cdot & \vdash & ('s, a) & ! & (x, y, z) & \searrow & ((\mathsf{let}\ x = {'s}\ \mathsf{in}\ x)\,, a) \\ a <: nat & \vdash & ('s, a) & ! & (x, y, z) & \searrow & x = {'s}, y = {'s}, z = nat \end{array}$$

3.5 Determinism

For each non-deterministic evaluation judgment $\searrow$, we define a deterministic counterpart $\searrow\!\!\searrow$, which collects all possible evaluation results into a sum.

$$\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \searrow\!\!\searrow \bigoplus\{w \mid \Gamma \vdash \llparenthesis e \rrparenthesis_\rho \searrow w\}$$

$$\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \searrow\!\!\searrow^{!} \bigoplus\{w \mid \Gamma \vdash \llparenthesis e \rrparenthesis_\rho \searrow^{!} w\}$$

$$\Gamma \vdash \llparenthesis e \rrparenthesis_\rho \searrow\!\!\searrow \bigoplus\{(w, w') \mid \Gamma \vdash \llparenthesis e \rrparenthesis_\rho \searrow w, w'\}$$

4 Type system

Note 7. *In the following, we will often encounter rules that round information either upward or downward. This is justified by transitivity of parthood, i. e.,*

$$\forall A, B, C \ \ (A <: B \land B <: C \implies A <: C)$$

from which arises the general rule that in order to stay sound, the formalism may only round the left-hand side of a subtyping relation to be checked upward*, while the right-hand side may only be rounded* downward.

In particular, the typing context Γ*, which stores known bounds of expressions, is applicable to the left-hand side of the subtype relationship only, while non-determinism may only be applied on the right-hand side.*

4.1 Basic checking rules

$\boxed{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow B}$ **Parthood.** The fundamental type-checking judgment $\Gamma \vdash (\!|e|\!)_\delta \Leftarrow B$ states that in the context Γ, e interpreted within the environment δ is guaranteed to be a mereological *part* (i. e., a subtype) of B. B is assumed to be a closed expression. The parameters occurring in e must have bounds provided by Γ. Parameters may independently occur in B irrespective of the presence of a bound in Γ.

Basic rules. Bound checking generally proceeds according to structural induction over the left-hand-side expression, as given by the following set of rules.

$$\frac{\Gamma \vdash (\!|\delta(x)|\!)_{\cdot} \Leftarrow A}{\Gamma \vdash (\!|x|\!)_\delta \Leftarrow A} \qquad \frac{\cdot \vdash C \searrow^{!} c}{\Gamma \vdash (\!|c|\!)_\delta \Leftarrow C} \qquad \frac{\cdot \vdash C \searrow^{!} a}{\Gamma \vdash (\!|a|\!)_\delta \Leftarrow C}$$

$$\frac{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow \top \qquad \Gamma \vdash (\!|e'|\!)_{\delta, x = (\mathsf{let}\ \delta\ \mathsf{in}\ e)} \Leftarrow A}{\Gamma \vdash (\!|\mathsf{let}\ x = e\ \mathsf{in}\ e'|\!)_\delta \Leftarrow A} \qquad \frac{\forall i.\ \Gamma \vdash (\!|e_i|\!)_\delta \Leftarrow C}{\Gamma \vdash (\!|\bigoplus \vec{e}|\!)_\delta \Leftarrow C}$$

$$\frac{\begin{array}{c}\forall i.\ \Gamma \vdash (\!|p_i|\!)_\delta \qquad \cdot \vdash A \searrow^{!} \mathsf{let}\ \rho\ \mathsf{in}\ \mathsf{fun}\ \{\overrightarrow{q \to t}\} \qquad \forall j.\ \Gamma \vdash (\!|q_j|\!)_\rho \lhd (\!|\vec{p}|\!)_\delta \\ \forall i, j.\ \Gamma \vdash (\!|p_i \to e_i|\!)_\delta <: (\!|q_j \to t_j|\!)_\rho\end{array}}{\Gamma \vdash (\!|\mathsf{fun}\ \{\overrightarrow{p \to e}\}|\!)_\delta \Leftarrow A}\ (\Leftarrow\mathsf{fun})$$

$$\frac{\Gamma, a <: (\mathsf{let}\ \delta\ \mathsf{in}\ A) \vdash (\!|e|\!)_{\delta, x = a} \Leftarrow A \qquad \Gamma \vdash (\!|A|\!)_\delta \Leftarrow C}{\Gamma \vdash (\!|\mathsf{fix}\ (x <: A) \to e|\!)_\delta \Leftarrow C}$$

$$\frac{\cdot \vdash C \searrow^{!} \mathsf{let}\ \rho\ \mathsf{in}\ (C_1, C_2) \qquad \Gamma \vdash (\!|e_1|\!)_\delta \Leftarrow \mathsf{let}\ \rho\ \mathsf{in}\ C_1 \qquad \Gamma \vdash (\!|e_2|\!)_\delta \Leftarrow \mathsf{let}\ \rho\ \mathsf{in}\ C_2}{\Gamma \vdash (\!|(e_1, e_2)|\!)_\delta \Leftarrow C}$$

$$\frac{\Gamma \vdash (\!|e|\!)_\delta \Lleftarrow (\mathsf{let}\ \delta\ \mathsf{in}\ t) \qquad \Gamma \vdash (\!|t|\!)_\delta \Lleftarrow C}{\Gamma \vdash (\!|(e <: t)|\!)_\delta \Lleftarrow C}$$

$$\frac{\Gamma \vdash (\!|f\ e|\!)_\delta @\cdot \Lleftarrow C}{\Gamma \vdash (\!|f\ e|\!)_\delta \Lleftarrow C}\ (\Lleftarrow app) \qquad \frac{\Gamma \vdash (\!|e|\!)_\delta\ !!\ p}{\Gamma \vdash (\!|e|\!)_\delta \Lleftarrow \{p\}}\ (\Lleftarrow cmp)$$

Most rules should be self-explanatory. The rules $(\Lleftarrow fun)$, $(\Lleftarrow app)$, and $(\Lleftarrow cmp)$, however, do warrant an explanation.

$(\Lleftarrow cmp)$ In order to verify that a value v belongs to a comprehension $\{p\}$, we need to prove that v matches the pattern p. Since we need to do this statically for an expression e, a great deal of machinery is necessary to do case distinctions on the known type (i. e., bound) of e. See Section 4.7 for details.

$(\Lleftarrow app)$ In a way, application checking is the heart of every type checker. Here, too, the complexity of the task warrants its own judgment. See Section 4.2 for details.

$(\Lleftarrow fun)$ Function subtyping, finally, is refreshingly simple. To prove that $\mathsf{fun}\ \{\overrightarrow{p \to e}\} <: \mathsf{fun}\ \{\overrightarrow{q \to t}\}$, we need only check two things:

1. Argument contravariance: Any value matching some q_j must also match some p_i. This is *pattern coverage*, which is described in Section 4.12.
2. Result type covariance: Under the assumption that p_i and q_j simultaneously match a value v, e_i must be a subtype of t_j after matching. This is *clause compatibility*, covered in Section 4.8.

Rounding. As noted in the beginning of this section, left-hand-side expressions may always be rounded toward their bounds as stored in the bound context Γ. We introduce a straight-forward rule to this effect.

$$\frac{\Gamma \vdash (\!|\Gamma(a)|\!)_\cdot \Lleftarrow C}{\Gamma \vdash (\!|a|\!)_\delta \Lleftarrow C}$$

Well-formedness. While the top type $\top$ is technically a special case of a comprehension ($\top = \{x\}$), special rules must still be provided for well-formedness checking in order to avoid the need for circular reasoning.

$$\frac{\forall i.\ \Gamma \vdash \llparenthesis p_i \rrparenthesis_\delta \qquad \forall i.\ \Gamma \vdash \llparenthesis p_i \to e_i \rrparenthesis_\delta <: \llparenthesis \_ \to \top \rrparenthesis_\rho}{\Gamma \vdash \llparenthesis \mathsf{fun}\ \{\overrightarrow{p \to e}\} \rrparenthesis_\delta \Leftarrow \top}$$

$$\frac{}{\Gamma \vdash \llparenthesis c \rrparenthesis_\delta \Leftarrow \top} \qquad \frac{}{\Gamma \vdash \llparenthesis a \rrparenthesis_\delta \Leftarrow \top} \qquad \frac{\Gamma \vdash \llparenthesis e_1 \rrparenthesis_\delta \Leftarrow \top \qquad \Gamma \vdash \llparenthesis e_2 \rrparenthesis_\delta \Leftarrow \top}{\Gamma \vdash \llparenthesis (e_1, e_2) \rrparenthesis_\delta \Leftarrow \top}$$

Structural context checking. Finally, in order to be able to check general stuck expressions, we can do a structural check instead of rounding.

$$\frac{\cdot \vdash C \searrow^{!} E'[a] \qquad \Gamma \vdash E <: E'}{\Gamma \vdash E[a] <: C}$$

Example 8. *Assuming the definitions of Section 2.2, we can prove the conjecture made earlier:*

$$\frac{\dfrac{}{\vdash \llparenthesis ('s, n), (x, xs) \rrparenthesis_\cdot}(1) \quad \dfrac{}{\vdash \mathsf{fun}\ldots \searrow^{!} \mathsf{let}\ \cdot\ \mathsf{in}\ \mathsf{fun}\ldots}(2) \quad \dfrac{}{\vdash \llparenthesis\ldots\rrparenthesis_\cdot \lhd \llparenthesis\ldots\rrparenthesis_\cdot}(3) \quad \dfrac{}{\vdash \llparenthesis \ldots \to id\ a\ x \rrparenthesis_\cdot <: \llparenthesis \ldots \to a \rrparenthesis_\cdot}(4)}{\vdash \llparenthesis \mathsf{fun}\ \{(('s, n), (x, xs)) \to id\ a\ x\} \rrparenthesis_\cdot \Leftarrow \mathsf{fun}\ \{((m <: ('s, nat)), (v <: vec\ a\ m)) \to a\}}$$

Subderivations will be given as the corresponding judgments are introduced. They have been assigned reference numbers for this purpose.

Derivation (2)*, however, can be given immediately:*

$$\frac{}{\vdash \mathsf{fun}\ \{((m\ldots), (v\ldots)) \to id\ a\ x\} \searrow^{!} \mathsf{let}\ \cdot\ \mathsf{in}\ \mathsf{fun}\ \{((m\ldots), (v\ldots)) \to id\ a\ x\}}\ \text{by axiom}$$

4.2 Application checking.

$\boxed{\Gamma \vdash \llparenthesis f \rrparenthesis_\delta @ \vec{e} \Leftarrow C}$

Checking an application $f\ \vec{e}$ works by first collecting the arguments $\vec{e}$ from right to left as long as the applicand f is not a fun form, rounding and unfolding the applicand as necessary, and finally applying the collected arguments from left to right by pattern matching.

The collection and base rules are as follows.

$$\frac{\Gamma \vdash \llparenthesis e \rrparenthesis_{\delta} \Leftarrow C}{\Gamma \vdash \llparenthesis e \rrparenthesis_{\delta} @ \cdot \Leftarrow C}$$

$$\frac{\Gamma \vdash \llparenthesis f \rrparenthesis_{\delta} @ \big((\mathsf{let}\ \delta\ \mathsf{in}\ e)\,, \vec{e} \big) \Leftarrow C}{\Gamma \vdash \llparenthesis f\ e \rrparenthesis_{\delta} @ \vec{e} \Leftarrow C} \qquad \frac{\Gamma \vdash \llparenthesis e \rrparenthesis_{\delta} \Leftarrow \top \qquad \Gamma \vdash \llparenthesis e' \rrparenthesis_{\delta, x = (\mathsf{let}\ \delta\ \mathsf{in}\ e)} @ \vec{e} \Leftarrow C}{\Gamma \vdash \llparenthesis \mathsf{let}\ x = e\ \mathsf{in}\ e' \rrparenthesis_{\delta} @ \vec{e} \Leftarrow C}$$

$$\frac{\Gamma, a <: (\mathsf{let}\ \delta\ \mathsf{in}\ A) \vdash \llparenthesis f \rrparenthesis_{\delta, x = a} @ \vec{e} \Leftarrow A \qquad \Gamma \vdash \llparenthesis A \rrparenthesis_{\delta} \Leftarrow \top}{\Gamma \vdash \llparenthesis \mathsf{fix}\ (x <: A) \to f \rrparenthesis_{\delta} @ \vec{e} \Leftarrow C}\ (@fix)$$

$$\frac{\Gamma \vdash \llparenthesis \delta(x) \rrparenthesis_{.} @ \vec{e} \Leftarrow C}{\Gamma \vdash \llparenthesis x \rrparenthesis_{\delta} @ \vec{e} \Leftarrow C} \qquad \frac{\Gamma \vdash \llparenthesis \Gamma(a) \rrparenthesis_{.} @ \vec{e} \Leftarrow C}{\Gamma \vdash \llparenthesis a \rrparenthesis_{\delta} @ \vec{e} \Leftarrow C}$$

The actual application check can be performed by two competing rules.

$$\frac{\begin{array}{c} \Gamma \vdash \llparenthesis e_1 \rrparenthesis_{.} \Leftarrow \top \\ \Gamma \vdash e_1 \twoheadrightarrow \vec{e_1} \searrow \Gamma' \\ \forall i.\ \exists j.\ \Gamma' \vdash \llparenthesis e_1^i \rrparenthesis_{.} \Leftarrow \llparenthesis (p_j \to t_j) \rrparenthesis_{\delta} \searrow D; \Gamma'' \\ \Gamma'' \vdash \llparenthesis D \rrparenthesis_{.} @ (e_2, \ldots, e_n) \Leftarrow C \end{array}}{\Gamma \vdash \llparenthesis \mathsf{fun}\ \{\overrightarrow{p \to t}\} \rrparenthesis_{\delta} @ (e_1, \ldots, e_n) \Leftarrow C}\ (@check)$$

$$\frac{\begin{array}{c} \Gamma \vdash \llparenthesis (\mathsf{let}\ \delta\ \mathsf{in}\ f)\, e_1 \rrparenthesis_{.} \Leftarrow \top \\ \Gamma \vdash \llparenthesis (\mathsf{let}\ \delta\ \mathsf{in}\ f)\, e_1 \rrparenthesis_{.} \Searrow \vec{r} \\ \Gamma \vdash \llparenthesis \bigoplus \vec{r} \rrparenthesis_{.} @ e_2, \ldots, e_n \Leftarrow C \end{array}}{\Gamma \vdash \llparenthesis f \rrparenthesis_{\delta} @ e_1, \ldots, e_n \Leftarrow C}\ (@eval)$$

Explanation:

$(@check)$ This is the traditional application checking rule. For the applicand f, we enforce a function form $\mathsf{fun}\ \{\overrightarrow{p \to t}\}$. We first check that for all values that might be produced by the argument e_1, some p_i will match. Finally, for each matching case, we verify that the corresponding clause produces a subtype of whatever type we are checking against. See Section 4.5 for details on how we compute the type resulting from the match.

In order to be able to do the case distinction, we split e_1 into subcases $\vec{e_1}$ and check each subcase separately. Splitting is non-deterministic (see Section 4.12); there are many possible outcomes. A heuristic that works well for match checking is described in Section 5.6.1.

(*@eval*) Alternatively, we can simply check the application for well-formedness (which is a lot easier than checking that it produces the correct result type) and unfold it statically, at checking time.

Note 9. *The* (*@eval*) *rule is in obvious competition with the* (*@check*) *rule. Note that the* (*@check*) *rule actually suffices in the absence of polymorphic function types. In fact,* (*@eval*) *is precisely the rule to use when a function is applied to a "type" argument. Since there is no distinction between types and values in* TypeCore, *the ambiguity appears to be essential.*

An actual implementation might want to introduce explicit annotations to resolve the ambiguity between run-type and compile-time application. Our implementation does, in fact, provide such a mechanism. See Section 5.6.2 for details.

Example 10.

$$\dfrac{}{nn <: nat \vdash \llparenthesis('s, nn)\rrparenthesis\,.\, \leftleftarrows \llparenthesis((k <: nat) \to b)\rrparenthesis\,.\, \searrow b; nn <: nat}\,(\ldots)$$

$$\dfrac{\dfrac{}{nn <: nat \vdash \llparenthesis('s, nn)\rrparenthesis\,.\, \leftleftarrows \top}\,(\ldots) \qquad \dfrac{\dfrac{}{nn <: nat \vdash \llparenthesis b\rrparenthesis\,.\, \leftleftarrows \top}\,\text{axiom}}{nn <: nat \vdash \llparenthesis b\rrparenthesis\,.\,@ \leftleftarrows \top}}{nn <: nat \vdash \llparenthesis \mathsf{fun}\{(k <: nat) \to b\}\rrparenthesis\,.\,@('s, nn) \leftleftarrows \top}\,(@check)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{nn <: nat \vdash \llparenthesis a\rrparenthesis\,.\, \leftleftarrows \top}\,\text{axiom} \qquad \dfrac{}{\ldots \vdash \llparenthesis a\rrparenthesis\,.\, \leftleftarrows \llparenthesis(b \to \mathsf{fun}\{(k \ldots) \to b\})\rrparenthesis\,.\, \searrow \mathsf{fun}\{(k \ldots) \to a\}; \ldots}\,(\ldots)}{nn <: nat \vdash \llparenthesis \mathsf{fun}\{b \to \mathsf{fun}\{(k <: nat) \to b\}\}\rrparenthesis\,.\,@a, ('s, nn) \leftleftarrows \top}\,(@check)}{nn <: nat \vdash \llparenthesis vec\rrparenthesis\,.\,@a, ('s, nn) \leftleftarrows \top}\,(\text{note})}{nn <: nat \vdash \llparenthesis vec\ a\rrparenthesis\,.\,@('s, nn) \leftleftarrows \top}}{nn <: nat \vdash \llparenthesis vec\ a\ ('s, nn)\rrparenthesis\,.\,@ \leftleftarrows \top}}{(7) \iff nn <: nat \vdash \llparenthesis vec\ a\ ('s, nn)\rrparenthesis\,.\, \leftleftarrows \top}$$

Note 11. *In the example derivation, we round vec as if it were a parameter whose bound is contained in* Γ. *In the actual implementation, global variables are handled specially so as to make this possible. See Section 5.1.*

4.3 Context refinement

For structural checking of normals, we define two judgments that check evaluation and matching contexts for an inclusion relationship depending on their syntactic class.

$\boxed{\Gamma \vdash \llparenthesis E\rrparenthesis_\delta \leftleftarrows E'}$ **Evaluation context refinement.**

$$\frac{}{\Gamma \vdash (\!|\cdot|\!)_\delta \Leftarrow \cdot}$$

$$\frac{\Gamma \vdash (\!|E|\!)_\delta \Leftarrow E' \qquad \Gamma \vdash (\!|A|\!)_\delta \Leftarrow A'}{\Gamma \vdash (\!|E\ A|\!)_\delta \Leftarrow E'\ A'} \qquad \frac{\Gamma \vdash (\!|w_f|\!)_\delta \Leftarrow w'_f \qquad \Gamma \vdash (\!|S|\!)_\delta \Leftarrow S'}{\Gamma \vdash (\!|w_f\ S|\!)_\delta \Leftarrow w'_f\ S'}$$

$\boxed{\Gamma \vdash (\!|S|\!)_\delta \Leftarrow S'}$ **Matching context refinement.**

$$\frac{\Gamma \vdash (\!|S|\!)_\delta \Leftarrow S' \qquad \Gamma \vdash (\!|A|\!)_\delta \Leftarrow A'}{\Gamma \vdash (\!|(S, A)|\!)_\delta \Leftarrow (S', A')} \qquad \frac{\Gamma \vdash (\!|A|\!)_\delta \Leftarrow A' \qquad \Gamma \vdash (\!|S|\!)_\delta \Leftarrow S'}{\Gamma \vdash (\!|(A, S)|\!)_\delta \Leftarrow (A', S')}$$

Note 12 (Promotion vs. structural subtyping)**.** *When checking a stuck application, we need to choose between rounding the head towards its bound or doing a structural check. As the following example adapted from [Ste98, Ex. 5.93] demonstrates, structural checking is not always the right choice.*

$$\frac{\begin{gathered}\Gamma \vdash (\mathsf{fun}\ y \to y\ 1)\ (\mathsf{fun}\ x \to a\ (\mathsf{minus}\ x)) \searrow^{!} a\ (\mathsf{minus}\ 1) \\ \ldots \vdash a\ (\mathsf{minus}\ 1) <: a\ (\mathsf{minus}\ 1)\end{gathered}}{a <: \mathsf{fun}\ (y <: \mathsf{Nat} \to \mathsf{Nat}) \to y\ 1 \vdash a\ (\mathsf{fun}\ (x <: \mathsf{Nat}) \to a\ (\mathsf{minus}\ x)) <: a\ (\mathsf{minus}\ 1)}$$

4.4 Interlude: Pattern normalization

$\boxed{(\!|p|\!)_\delta \searrow^{\mathsf{P}} p'}$ **Pattern normalization.** Pattern normalization pushes an enviroment δ into the bounds of a pattern p, thereby closing p with regard to δ.

This is mainly a convenience predicate. We will often use it in the following inductive definitions of predicates to avoid having to explicitly pass environments along with every pattern.

$$\frac{}{(\!|x|\!)_\delta \searrow^{\mathsf{P}} x} \qquad \frac{}{(\!|c|\!)_\delta \searrow^{\mathsf{P}} c} \qquad \frac{(\!|p_1|\!)_\delta \searrow^{\mathsf{P}} p'_1 \qquad (\!|p_2|\!)_\delta \searrow^{\mathsf{P}} p'_2}{(\!|p_1, p_2|\!)_\delta \searrow^{\mathsf{P}} (p'_1, p'_2)}$$

$$\frac{(\!|p|\!)_\delta \searrow^{\mathsf{P}} p'}{(\!|p <: B|\!)_\delta \searrow^{\mathsf{P}} (p' <: \mathsf{let}\ \delta\ \mathsf{in}\ B)}$$

4.5 Function argument checking

$\boxed{\Gamma \vdash (\!|e|\!)_\delta \leftleftharpoons (\!|(p \to t)|\!)_\rho \searrow C}$ **Function argument check.** We define a simple judgment that:

- verifies an argument e as a valid argument to a function clause $p \to t$ as interpreted in their respective environments, and
- yields the result of applying the environment resulting from the pattern match of e against p to the right hand side of the clause.

Both of these goals are easily implemented by resorting to the generic pattern match judgment, which is described in Section 4.7.

$$\frac{(\!|p|\!)_\rho \searrow^{\mathsf{P}} p' \qquad \Gamma \vdash (\!|e|\!)_\delta \leftleftharpoons p' <: \top \searrow \Gamma'; \rho'}{\Gamma \vdash (\!|e|\!)_\delta \leftleftharpoons (\!|(p \to t)|\!)_\rho \searrow \mathsf{let}\ \rho'\ \mathsf{in}\ t; \Gamma'}$$

4.6 Interlude: Intersection

Some of the following rules require the computation of type intersections. We define two kinds of intersection, one rounding upward when in doubt, the other rounding downward.

For notational simplicity, we will assume that both arguments contain no let-expressions. (Generalizing the definition to let-expressions is an easy exercise.) In addition, we will assume that the implementation unfolds its arguments as necessary to distinguish the defining cases (after checking them for well-formedness, if necessary). The common core is:

$$\begin{array}{rcllll}
lit_1 & \cap_e & lit_2 & := & lit_1 & \text{if } lit_1 = lit_2 \\
lit_1 & \cap_e & lit_2 & := & \bot & \text{if } lit_1 \neq lit_2 \\
lit & \cap_e & (\_,\_) & := & \bot & \\
(\_,\_) & \cap_e & lit & := & \bot & \\
A & \cap_e & \{x\} & := & A & \\
\{x\} & \cap_e & B & := & B & \\
(s_1, s_2) & \cap_e & (t_1, t_2) & := & (s_1 \cap_e t_1, s_2 \cap_e t_2) & \\
A & \cap_e & (B <: \_) & := & A \cap_e B & \\
(A <: \_) & \cap_e & B & := & A \cap_e B & \\
\left(\bigoplus_{i=1}^{n} A_i\right) & \cap_e & B & := & \bigoplus_{i=1}^{n} (A_i \cap_e B) & \\
A & \cap_e & \left(\bigoplus_{i=1}^{n} B_i\right) & := & \bigoplus_{i=1}^{n} (A \cap_e B_i) & \\
A & \cap_e & B & := & e & \text{if no other clause matches}
\end{array}$$

This common core is then used for both upward- and downward-rounding intersection:

$$\begin{aligned} \overline{\sqcap} &:= \sqcap_{\top} \\ \underline{\sqcap} &:= \sqcap_{\bot} \end{aligned}$$

4.7 Pattern matching

$\boxed{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow p <: B \searrow \Gamma'; \rho}$ **Static pattern match**. The judgment defined below tries to match an expression e against a pattern p. If successful, the result is an environment ρ assigning the pattern variables of p to the corresponding subexpressions of e.

The bound B is used to refute matches that do not satisfy the bound conditions asserted by the pattern p.

Note that the bound needs to be split into a left and right part in the pair case. This is done by the use of a special predicate designed for this purpose. See Section 4.11 for details.

Note 13. *Since pair splitting is imprecise, it can lose useful information. This can prevent the refutation of matches that would otherwise be refutable.*

$$\frac{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow B}{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow x <: B \searrow \Gamma; (x = \mathsf{let}\ \delta\ \mathsf{in}\ e)} \qquad \frac{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow c \,\underline{\sqcap}\, B}{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow c <: B \searrow \Gamma; \cdot}$$

$$\frac{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow p <: A \,\underline{\sqcap}\, B \searrow \Gamma'; \rho}{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow (p <: A) <: B \searrow \Gamma'; \rho}$$

$$\frac{\begin{array}{c} \Gamma \vdash B \prec B_1, B_2 \searrow \Gamma_1 \\ \Gamma_1 \vdash (\!|e_1|\!)_\delta \Leftarrow p_1 <: B_1 \searrow \Gamma_2; \rho_1 \\ (\!|p_2|\!)_{\rho_1} \overset{\mathsf{P}}{\searrow} p_2' \\ \Gamma_2 \vdash (\!|e_2|\!)_\delta \Leftarrow p_2' <: B_2 \searrow \Gamma_3; \rho_2 \end{array}}{\Gamma \vdash (\!|e_1, e_2|\!)_\delta \Leftarrow (p_1, p_2) <: B \searrow \Gamma_3; (\rho_1, \rho_2)}$$

$$\frac{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow B \qquad \Gamma \vdash (\!|e|\!)_\delta \searrow v \qquad \Gamma \vdash (\!|v|\!).\ !\ p \searrow \rho}{\Gamma \vdash (\!|e|\!)_\delta \Leftarrow p <: B \searrow \Gamma; \rho}\ p = (p_1, p_2)$$

The pattern matching rules are relatively straight-forward. One thing to notice is the need for computing intersections as defined in Section 4.6.

$\boxed{\Gamma \vdash \llparenthesis e \rrparenthesis_\delta \;!!\; p}$ **Pattern match with case distinction.** While the static pattern match judgment directly matches an expression e with a pattern p, we often need to split e into more basic cases in order for the pattern match to succeed. In particular, this can be necessary when p contains dependent pair patterns that need more precise information for the unfolding of type bounds than provided by e in the general case.

To fulfil this need, we define a pattern match judgment with built-in type splitting (Section 4.12) as follows.

$$\frac{\Gamma \vdash \llparenthesis e \rrparenthesis_\delta \leftleftarrows \top \qquad \Gamma \vdash (\mathsf{let}\ \delta\ \mathsf{in}\ e) \twoheadrightarrow \vec{e} \searrow \Gamma' \qquad \forall i.\ \Gamma' \vdash \llparenthesis e_i \rrparenthesis_{\cdot} \;!!\; p}{\Gamma \vdash \llparenthesis e \rrparenthesis_\delta \;!!\; p}\;(EMsplit)$$

$$\frac{\Gamma \vdash \llparenthesis e \rrparenthesis_\delta \leftleftarrows p <: \top \searrow \_;\_}{\Gamma \vdash \llparenthesis e \rrparenthesis_\delta \;!!\; p}\;(EMprim)$$

Example 14.

$$\frac{}{nn <: nat \vdash \llparenthesis vec\ a\ ('s, nn) \rrparenthesis_{\cdot} \searrow (a, vec\ a\ nn)}\;(\ldots)$$

$$\dfrac{\dfrac{\dfrac{}{nn <: nat \vdash \llparenthesis vec\ a\ ('s, nn) \rrparenthesis_{\cdot} \leftleftarrows \top}\;(7) \qquad \dfrac{}{nn <: nat \vdash \llparenthesis (a, vec\ a\ nn) \rrparenthesis_{\cdot}\ !\ (x, xs) \searrow \_}\;(\ldots)}{nn <: nat \vdash \llparenthesis vec\ a\ ('s, nn) \rrparenthesis_{\cdot} \leftleftarrows (x, xs) <: \top \searrow \_;\_}}{(5) \iff nn <: nat \vdash \llparenthesis vec\ a\ ('s, nn) \rrparenthesis_{\cdot} \;!!\; (x, xs)}$$

Example 15.

$$\dfrac{\dfrac{\dfrac{}{\vdash \top \prec \top, \top \searrow \cdot}\;(\ldots) \quad \dfrac{\dfrac{}{\vdash \llparenthesis 's \rrparenthesis_{\cdot} \leftleftarrows 's}\;\text{axiom}}{\vdash \llparenthesis 's \rrparenthesis_{\cdot} \leftleftarrows' s <: \top \searrow \cdot; \cdot}\;{}'s \sqcap \top = {}'s \quad \dfrac{\dfrac{}{\vdash \llparenthesis nat \rrparenthesis_{\cdot} \leftleftarrows \top}\;\text{axiom}}{\vdash \llparenthesis nat \rrparenthesis_{\cdot} \leftleftarrows n <: \top \searrow \_;\_}}{\vdash \llparenthesis ('s, nat) \rrparenthesis_{\cdot} \leftleftarrows ('s, n) <: \top \searrow \_;\_}}{(6) \iff \vdash \llparenthesis ('s, nat) \rrparenthesis_{\cdot} \;!!\; ('s, n)}$$

$\boxed{\Gamma \vdash \llparenthesis p \rrparenthesis_\delta}$ **Pattern well-formedness.** A pattern p is well-formed if it is linear and all its bounds are well-formed. The latter is defined in terms of the bound-extraction judgment (see Section 4.10).

$$\Gamma \vdash \llparenthesis p \rrparenthesis_\delta \iff p \text{ linear and } \exists \Gamma', \rho.\ \Gamma \vdash p <: \top \searrow \Gamma'; \rho$$

Example 16.

$$\dfrac{\dfrac{}{nn <: \top \vdash x <: \top \searrow nn <: \top, xx <: \top; n = nn, x = xx} \quad \dfrac{}{nn <: \top, xx <: \top \vdash xs <: \top \searrow \_;\_}}{nn <: \top \vdash (x, xs) <: \top \searrow \_;\_}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash' s <: \top \searrow \cdot;\cdot} \quad \dfrac{}{\vdash n <: \top \searrow nn <: \top; n = nn}}{\vdash ('s, n) <: \top \searrow nn <: \top; n = nn} \quad \vdots}{\vdash ('s, n), (x, xs) <: \top \searrow \_;\_}}{(1) \iff \vdash \llparenthesis ('s, n), (x, xs) \rrparenthesis_{\cdot}}\ ('s,n),(x,xs) \text{ linear}$$

4.8 Function clause compatibility

$\boxed{\Gamma \vdash \llparenthesis p \to e \rrparenthesis_\delta <: \llparenthesis q \to t \rrparenthesis_\rho}$ **Clause compatibility.** We consider a function clause $p \to e$ to be compatible with the clause $q \to t$ iff for all values that simultaneously match both p and q, e is a subtype of t after the corresponding pattern match.

In particular, if p and q are disjoint, i. e., no value can ever simultaneously match both p and q, the clauses are considered compatible.

Whether simultaneous matching of patterns p and q is possible, and if so, what the outcome is, is determined by unification. See Section 4.9 for details.

Formally:

$$\dfrac{\begin{array}{c}\llparenthesis p \rrparenthesis_\delta \overset{\mathsf{P}}{\searrow} p_\delta \\ \llparenthesis q \rrparenthesis_\rho \overset{\mathsf{P}}{\searrow} q_\rho \\ \forall \Gamma', \delta', \rho' \left(\Gamma \vdash p_\delta \doteq q_\rho <: \top \searrow \Gamma', \delta', \rho' \implies \Gamma' \vdash \llparenthesis e \rrparenthesis_{\delta,\delta'} \leftleftarrows \mathsf{let}\ \rho, \rho'\ \mathsf{in}\ t \right)\end{array}}{\Gamma \vdash \llparenthesis p \to e \rrparenthesis_\delta <: \llparenthesis q \to t \rrparenthesis_\rho}$$

Example 17.

$$\dfrac{}{xx <: a, nn <: nat, \ldots \vdash \llparenthesis x \rrparenthesis_{x=xx} \leftleftarrows a}\ (\ldots)$$

$$\dfrac{\dfrac{}{\vdash (('s, n), (x, xs)) \doteq ((m <: ('s, nat)), (v <: vec\ a\ m)) <: \top \searrow \ldots;\ldots;\ldots}\ (8) \quad \vdots}{(4) \iff \vdash \llparenthesis (('s, n), (x, xs)) \to x \rrparenthesis_{\cdot} <: \llparenthesis ((m <: ('s, nat)), (v <: vec\ a\ m)) \to a \rrparenthesis_{\cdot}}$$

4.9 Unification

$\boxed{\Gamma \vdash p \doteq q <: B \searrow \Gamma'; \delta; \rho}$ **Pattern unification.** The following set of rules defines unification of patterns p and q for values bounded by B. The result is a pair $(\delta; \rho)$ of environments that bind all pattern variables of p and q, respectively, possibly establishing dependencies between the LHS and RHS by binding the same set of parameters to variables on either side.

Pattern bounds contained in p and q are used to augment the context Γ' with bound information about newly generated parameters. Bounds are rounded upwards if necessary.

In the base case, unification relies on bound extraction (see Section 4.10) to determine variable bounds to be stored in Γ'.

Unification fails iff it can prove that no value can ever satisfy all of p, q, and B simultaneously.

Note 18. *In the worst case, unification can determine neither incompability between p and q nor any useful type bounds. In this case, Γ' will contain $\top$ as the bound for every variable in p and q.*

It is never the case that unification fails because of limited knowledge. Failure can only occur if p and q can be proven *incompatible.*

$$\frac{c \mathbin{\overline{\cap}} B \neq \{\}}{\Gamma \vdash c \doteq c <: B \searrow \Gamma; \cdot; \cdot}$$

$$\frac{\Gamma \vdash q <: B \searrow \Gamma'; \rho'}{\Gamma \vdash x \doteq q <: B \searrow \Gamma'; (x = \mathsf{let}\ \rho'\ \mathsf{in}\ q); \rho'}$$

$$\frac{\Gamma \vdash p <: B \searrow \Gamma'; \delta'}{\Gamma \vdash p \doteq y <: B \searrow \Gamma'; \delta'; (y = \mathsf{let}\ \delta'\ \mathsf{in}\ p)}$$

$$\frac{\Gamma \vdash p \doteq q <: A \mathbin{\overline{\cap}} B \searrow \Gamma'; \delta'; \rho'}{\Gamma \vdash p <: A \doteq q <: B \searrow \Gamma'; \delta'; \rho'}$$

$$\frac{\Gamma \vdash p \doteq q <: A \mathbin{\overline{\cap}} B \searrow \Gamma'; \delta'; \rho'}{\Gamma \vdash p \doteq q <: A <: B \searrow \Gamma'; \delta'; \rho'}$$

$$\frac{\begin{array}{c}\cdot \vdash (\!|B|\!). \searrow\!\!\searrow^{!} \vec{B} \qquad \vec{B}' = \{B_i | B_i \neq s \wedge B_i \neq \mathsf{let}\ \rho\ \mathsf{in}\ \mathsf{fun}\{...\}\} \\ \Gamma \vdash p_1, p_2 \doteq q_1, q_2 <: \bigoplus \vec{B}' \searrow \Gamma'; \delta; \rho\end{array}}{\Gamma \vdash p_1, p_2 \doteq q_1, q_2 <: B \searrow \Gamma'; \delta; \rho}$$

$$\frac{\begin{array}{c}\Gamma \vdash p_1 \doteq q_1 <: (\mathsf{let}\ \beta\ \mathsf{in}\ B_1) \searrow \Gamma_1; \delta_1; \rho_1 \\ (\!|p_2|\!)_{\delta_1} \searrow^{\mathsf{P}} p_2' \qquad (\!|q_2|\!)_{\rho_1} \searrow^{\mathsf{P}} q_2' \\ \Gamma_1 \vdash p_2' \doteq q_2' <: (\mathsf{let}\ \beta\ \mathsf{in}\ B_2) \searrow \Gamma_2; \delta_2; \rho_2\end{array}}{\Gamma \vdash p_1, p_2 \doteq q_1, q_2 <: \big(\mathsf{let}\ \beta\ \mathsf{in}\ (B_1, B_2)\big) \searrow \Gamma_2; (\delta_1, \delta_2); (\rho_1, \rho_2)}$$

$$\frac{\Gamma \vdash p \doteq q <: \top, \top \searrow \Gamma'; \delta'}{\Gamma \vdash p \doteq q <: B \searrow \Gamma'; \delta'}$$

Example 19.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash ('s,n) <: ('s,nat) \searrow nn <: nat, n = nn}\;(\ldots)}{\vdash ('s,n) \doteq m <: ('s,nat) \searrow nn <: nat; n = nn; m = ('s,nn)}}{\vdash ('s,n) \doteq (m <: ('s,nat)) <: \top \searrow nn <: nat; n = nn; m = ('s,nn)} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{nn <: nat \vdash (x,xs) <: (a, vec\ a\ nn) \searrow nn <: nat, xx <: a, xsxs <: vec\ a\ nn; v = (xx, xsxs)}\;(\ldots) \quad \dfrac{}{\vdash \llparenthesis vec\ a\ ('s,nn) \rrparenthesis .\ \searrow\!\!\searrow^{!} (a, vec\ a\ nn)}\;(\ldots)}{nn <: nat \vdash (x,xs) <: vec\ a\ ('s,nn) \searrow \ldots}}{nn <: nat \vdash (x,xs) \doteq v <: vec\ a\ ('s,nn) \searrow \ldots}}{nn <: nat \vdash (x,xs) \doteq (v <: vec\ a\ ('s,nn)) <: \top \searrow \ldots}}{\vdash (('s,n),(x,xs)) \doteq ((m <: ('s,nat)),(v <: vec\ a\ m)) <: (\top,\top) \searrow \ldots}}{(8) \iff \vdash (('s,n),(x,xs)) \doteq ((m <: ('s,nat)),(v <: vec\ a\ m)) <: \top \searrow xx <: a, \ldots; x = xx, \ldots; \ldots}
$$

4.10 Bound extraction

$\boxed{\Gamma \vdash p <: B \searrow \Gamma'; \delta}$ **Bound extraction.** The bound-extraction relation computes an environment binding all variables of the pattern p to newly generated parameters, whose bounds are determined by the type B. Bound computation rounds upwards, possibly losing information along the way, but may fail if it can prove that the pattern and bound are irreconciliable.

$$
\dfrac{}{\Gamma \vdash x <: B \searrow (\Gamma, a <: B); (x = a)} \qquad \dfrac{c \mathbin{\overline{\sqcap}} B \neq \{\}}{\Gamma \vdash c <: B \searrow \Gamma; \cdot}
$$

$$
\dfrac{\Gamma \vdash p <: A \mathbin{\overline{\sqcap}} B \searrow \Gamma'; \delta}{\Gamma \vdash p <: A <: B \searrow \Gamma'; \delta}
$$

$$
\dfrac{\cdot \vdash \llparenthesis B \rrparenthesis .\ \searrow\!\!\searrow^{!} \vec{B} \qquad \vec{B}' = \{B_i | B_i \neq s \wedge B_i \neq \mathsf{let}\ \rho\ \mathsf{in}\ \mathsf{fun}\{...\}\} \\ \Gamma \vdash p_1, p_2 <: \bigoplus \vec{B}' \searrow \Gamma_2; (\delta_1, \delta_2)}{\Gamma \vdash p_1, p_2 <: B \searrow \Gamma_2; (\delta_1, \delta_2)}\;(extrpairforget)
$$

$$
\dfrac{\Gamma \vdash p_1 <: \mathsf{let}\ \rho\ \mathsf{in}\ B_1 \searrow \Gamma_1; \delta_1 \qquad \llparenthesis p_2 \rrparenthesis_{\delta_1} \searrow^{\mathsf{P}} p_2' \qquad \Gamma_1 \vdash p_2' <: \mathsf{let}\ \rho\ \mathsf{in}\ B_2 \searrow \Gamma_2; \delta_2}{\Gamma \vdash p_1, p_2 <: \mathsf{let}\ \rho\ \mathsf{in}\ (B_1, B_2) \searrow \Gamma_2; (\delta_1, \delta_2)}
$$

$$
\dfrac{\llparenthesis q \rrparenthesis_{\rho} \searrow^{\mathsf{P}} q' \qquad q'' = renameVars(q') \qquad \Gamma \vdash p \doteq q'' <: \top \searrow \Gamma'; (\delta_1; \delta_2)}{\Gamma \vdash p_1, p_2 <: \mathsf{let}\ \rho\ \mathsf{in}\ \{q\} \searrow \Gamma'; (\delta_1, \delta_2)}
$$

$$\frac{\Gamma \vdash p_1, p_2 <: \top, \top \searrow \Gamma'; \delta}{\Gamma \vdash p_1, p_2 <: B \searrow \Gamma'; \delta}$$

Note 20. *Notice the peculiar base rule for pair patterns, $(extrpairforget)$. Since it throws away some information from B, it is not clear at first glance that it is sound (if you do not immediately see the danger in this, consider that we are now rounding a bound* downward*!). But note that the only information it throws away is about types that cannot possibly match a pair pattern and so cannot contribute to any bindings resulting from a pattern match. To see why this is useful, consider the case $(x, y) <: nat$. Without selectively discarding "bad" information, we cannot infer that in case the pattern matches, x will be 's, and y will be some value of type nat. Hence, a proposition as seemingly simple as* fun $\{'zero \rightarrow 'zero; ('s, n) \rightarrow ('s, n)\} <:$ fun $\{(x <: nat) \rightarrow nat\}$ *would not be provable because the bound for n would be inferred as being just $\top$.*

4.11 Pair Split

$\boxed{\Gamma \vdash v \prec v_1, v_2 \searrow \Gamma'}$ **Pair split.** The pair-split relation splits a closed expression v into a pair of closed expressions by case distinction across the structure of v. Pair splitting is a best-effort process that may round upwards (but not downwards); if an appropriate split is not possible, pair splitting returns $\top, \top$.

$$\frac{}{\Gamma \vdash (\mathsf{let}\ \rho\ \mathsf{in}\ (e_1, e_2)) \prec (\mathsf{let}\ \rho\ \mathsf{in}\ w_1), (\mathsf{let}\ \rho\ \mathsf{in}\ w_2) \searrow \Gamma}$$

$$\frac{(\!|p_1|\!)_\rho \overset{\mathsf{P}}{\searrow} p_1' \qquad \Gamma \vdash p_1' <: \top \searrow \Gamma_1; \rho_1}{\Gamma \vdash (\mathsf{let}\ \rho\ \mathsf{in}\ \{(p_1, p_2)\}) \prec (\mathsf{let}\ \rho\ \mathsf{in}\ \{p_1\}), (\mathsf{let}\ \rho, \rho_1\ \mathsf{in}\ \{p_2\}) \searrow \Gamma_1}$$

$$\frac{\Gamma \vdash B \prec B_1, B_2 \searrow \Gamma'}{\Gamma \vdash \{p <: B\} \prec B_1, B_2 \searrow \Gamma'} \qquad \frac{\Gamma \vdash A \overset{!}{\searrow} \_}{\Gamma \vdash A \prec \top, \top \searrow \Gamma}$$

Note 21. *Pair-splitting a comprehension destroys dependencies between the two components.*

4.12 Pattern coverage

$\boxed{\Gamma \vdash (\!|p|\!)_\delta \lhd (\!|\vec{q}|\!)_\rho}$ **Open pattern coverage.** The pattern coverage judgment determines whether a pattern p with bounds closed by an environment δ is covered by a set $\vec{q}$ of patterns with bounds closed by an environment ρ.

This is a convenience predicate; the actual coverage check is done by the closed pattern coverage predicate, which is defined below.

$$\frac{(\!|p|\!)_\delta \searrow^{\mathsf{P}} p' \qquad \forall i.\ (\!|q_i|\!)_\rho \searrow^{\mathsf{P}} q_i' \qquad \Gamma \vdash p' \lhd \vec{q}'}{\Gamma \vdash (\!|p|\!)_\delta \lhd (\!|\vec{q}|\!)_\rho}$$

$\boxed{\Gamma \vdash p \lhd \vec{q}}$ **Closed pattern coverage.** The closed pattern coverage judgment determines whether a pattern p is covered by a set $\vec{q}$ of patterns. All bounds occurring in p or $\vec{q}$ are assumed to be closed expressions.

$$\frac{\Gamma \vdash p \twoheadrightarrow \vec{p} \searrow \Gamma' \qquad \forall i.\ \Gamma' \vdash p_i \lhd \vec{q}}{\Gamma \vdash p \lhd \vec{q}}\ (PCsplit)$$

$$\frac{\Gamma \vdash p <: \top \longrightarrow_{x/o} p' \searrow \Gamma' \qquad \Gamma' \vdash p' \lhd \vec{q}}{\Gamma \vdash p \lhd \vec{q}}\ (PCprop)$$

$$\frac{\exists i.\ \Gamma \vdash p \lhd q_i \searrow \_;\_;\_}{\Gamma \vdash p \lhd \vec{q}}\ (PCprim)$$

Explanation:

$(PCsplit)$ This rule splits the pattern p into a set of patterns $\vec{p}$ that represent an exhaustive set of more primitive cases that p can be split into. See Section 4.12 for details on how patterns are split into subcases.

Example 22.

$$\vdash (x <: nat) \twoheadrightarrow \big(x <: ('s, nat)\big), (x <: 0) \searrow \cdot$$

$(PCprop)$ This rule lifts simple pattern bounds into patterns. This can be necessary after an application of the $(PCsplit)$ rule in order to further process the pattern effectively.

Example 23.

$$\begin{array}{lll} \big(x <: ('s, nat)\big) & \longrightarrow_{x/((x_1 <: 's),(x_2 <: nat))} & \big((x_1 <: 's), (x_2 <: nat)\big) \\ & \longrightarrow_{x_1/'s} & \big('s, (x_2 <: nat)\big) \end{array}$$

$(PCprim)$ This rule is singleton coverage, defined below.

$\boxed{\Gamma \vdash p \lhd q \searrow \Gamma'; \delta; \rho}$ **Singleton coverage.** Singleton coverage ensures that a pattern p is covered by a pattern q. The result is a pair $\delta; \rho$ of environments establishing bindings of the variables of the patterns p and q, respectively, to newly parameters bounded by the explicit bounds occurring in the patterns.

p is assumed to be of the form $p ::= c \mid (x <: A) \mid (p_1, p_2)$, i. e., bounds appear only around variables. This is ensured by sufficiently many iterations of pattern splitting (see Section 4.12), which is thereby assumed to have taken place sometime before this predicate is called upon.

$$\frac{}{\Gamma \vdash c \lhd c \searrow \Gamma; \cdot; \cdot} \qquad \frac{\Gamma \vdash \llparenthesis A \rrparenthesis .\ !!\, q \quad \Gamma \vdash q <: A \searrow \Gamma'; \rho}{\Gamma \vdash (x <: A) \lhd q \searrow \Gamma'; (x = \mathsf{let}\ \rho\ \mathsf{in}\ q); \rho}$$

$$\frac{\Gamma \vdash p <: \top \searrow \Gamma'; \delta}{\Gamma \vdash p \lhd y \searrow \Gamma'; \delta; (y = \mathsf{let}\ \delta\ \mathsf{in}\ p)}$$

$$\frac{\Gamma \vdash p \lhd A \searrow \_ \qquad \Gamma \vdash p \lhd q \searrow \Gamma'; \delta; \rho}{\Gamma \vdash p \lhd (q <: A) \searrow \Gamma'; \delta; \rho}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash p_1 \lhd q_1 \searrow \Gamma_2; \delta_1; \rho_1 \qquad \llparenthesis p_2 \rrparenthesis_{\delta_1} \overset{\mathsf{P}}{\searrow} p_2' \qquad \llparenthesis q_2 \rrparenthesis_{\rho_1} \overset{\mathsf{P}}{\searrow} q_2' \\ \Gamma_2 \vdash p_2' \lhd q_2' \searrow \Gamma_3; \delta_2; \rho_2\end{array}}{\Gamma \vdash (p_1, p_2) \lhd (q_1, q_2) \searrow \Gamma_3; (\delta_1, \delta_2); (\rho_1, \rho_2)}$$

$\boxed{\Gamma \vdash p \lhd C \searrow \Gamma'}$ **Pattern coverage by type.** Pattern coverage by type ensures that all values matching a pattern p are included in a given type C.

$$\frac{\Gamma \vdash \llparenthesis s \rrparenthesis .\ \Leftarrow C}{\Gamma \vdash s \lhd C \searrow \Gamma} \qquad \frac{\Gamma \vdash \llparenthesis \top \rrparenthesis .\ \Leftarrow C}{\Gamma \vdash x \lhd C \searrow \Gamma} \qquad \frac{\Gamma \vdash \llparenthesis A \rrparenthesis .\ \Leftarrow C}{\Gamma \vdash (p <: A) \lhd C \searrow \Gamma}$$

$$\frac{\vdash \llparenthesis C \rrparenthesis .\ \searrow\!\!\searrow (\mathsf{let}\ \rho\ \mathsf{in}\ \{q\}) \qquad \llparenthesis p \rrparenthesis_{\rho} \overset{\mathsf{P}}{\searrow} p' \qquad \Gamma \vdash p' \lhd q \searrow \Gamma'; \_; \_}{\Gamma \vdash p \lhd C \searrow \Gamma'}$$

$$\frac{\Gamma \vdash \{p\} \prec A_1, A_2 \searrow \Gamma' \qquad \Gamma' \vdash \llparenthesis (A_1, A_2) \rrparenthesis .\ \Leftarrow C}{\Gamma \vdash p \lhd C \searrow \Gamma'}$$

Note 24. *Pattern coverage by type and pattern coverage by pattern call upon each other: Pattern coverage calls upon type coverage when it encounters a type bound on the right-hand side; type coverage, in turn, calls upon pattern coverage when it encounters a comprehension. Care needs to be taken to ensure that no infinite loops occur between the two predicates.*

Example 25.

$$
\dfrac{
\dfrac{
\dfrac{}{\vdash (\!|('s,nat)|\!)_{.}\ !!\ ('s,n)}\,(6)
\quad
\dfrac{}{\vdash ('s,n) <: ('s,nat) \searrow nn <: nat; n = nn}\,(\ldots)
}{\vdash m <: ('s,nat) \lhd ('s,n) \searrow nn <: nat; m = ('s,nn); n = nn}
\quad
\dfrac{
\dfrac{}{nn <: nat \vdash (\!|vec\ a\ ('s,nn)|\!)_{.}\ !!\ (x,xs)}\,(5)
\quad
\dfrac{}{nn <: nat \vdash (x,xs) <: vec\ a\ ('s,nn) \searrow \_;\_}\,(\ldots)
}{nn <: nat \vdash (v <: vec\ a\ ('s,nn)) \lhd (x,xs) \searrow \_;\_;\_}
}{
\dfrac{\vdash ((m <: ('s,nat)),(v <: vec\ a\ m)) \lhd ((('s,n),(x,xs)) \searrow \_;\_;\_}{
\dfrac{\vdash ((m <: ('s,nat)),(v <: vec\ a\ m)) \lhd ((('s,n),(x,xs))}{(3) \iff \vdash (\!|((m <: ('s,nat)),(v <: vec\ a\ m))|\!)_{.} \lhd (\!|(('s,n),(x,xs))|\!)_{.}}
}
}
$$

$\boxed{\Gamma \vdash p \twoheadrightarrow \vec{p} \searrow \Gamma'}$ **Pattern split.** The pattern-split relation splits a pattern p into more primitive subcases $\vec{p}$ that cover the original pattern. This is necessary for coverage and matching checks that require case distinctions.

$$
\dfrac{}{\Gamma \vdash c \twoheadrightarrow c \searrow \Gamma}
\qquad
\dfrac{}{\Gamma \vdash x \twoheadrightarrow x \searrow \Gamma}
\qquad
\dfrac{\Gamma \vdash p \twoheadrightarrow \vec{p} \searrow \Gamma' \qquad \Gamma' \vdash A \twoheadrightarrow \vec{A} \searrow \Gamma''}{\Gamma \vdash (p <: A) \twoheadrightarrow \{(p_i <: A_j)\}_{i,j} \searrow \Gamma''}
$$

$$
\dfrac{\Gamma \vdash p \twoheadrightarrow p_1, \ldots, p_n \searrow \Gamma'}{\Gamma \vdash (p,q) \twoheadrightarrow (p_1,q), \ldots, (p_n,q) \searrow \Gamma'}
$$

$$
\dfrac{\Gamma \vdash p <: \top \searrow \Gamma_1; \delta \qquad (\!|q|\!)_\delta \searrow^{\mathsf{P}} q' \qquad \Gamma_1 \vdash q' \twoheadrightarrow q_1, \ldots, q_n \searrow \Gamma_2 \qquad \forall i.\ (\!|q_i|\!)_{\delta^{-1}} \searrow^{\mathsf{P}} q_i'}{\Gamma \vdash (p,q) \twoheadrightarrow (p,q_1'), \ldots, (p,q_n') \searrow \Gamma_2}
$$

$\boxed{\Gamma \vdash B \twoheadrightarrow \vec{B} \searrow \Gamma'}$ **Type split.** The type-split relation splits a type B into a set $\vec{B}$ of more primitive types. As with pattern splitting, this enables case distinctions where necessary.

$$
\dfrac{}{\Gamma \vdash e \twoheadrightarrow e \searrow \Gamma}
$$

$$
\dfrac{\Gamma_0 \vdash B \searrow\!\!\searrow^{!} B_1 \oplus \cdots \oplus B_n \qquad \forall i.\ \left(\Gamma_{i-1} \vdash B_i \twoheadrightarrow B_i^1, \ldots, B_i^{m_i} \searrow \Gamma_i\right)}{\Gamma_0 \vdash B \twoheadrightarrow \bigcup_{i=1}^{n} \bigcup_{k=1}^{m_i} \{B_i^k\} \searrow \Gamma_n}
$$

$$\frac{\Gamma \vdash B \searrow\!\!\searrow^{!} \mathsf{let}\ \rho\ \mathsf{in}\ \{p\} \qquad (\!|p|\!)_\rho \searrow^{\mathsf{P}} q \qquad \Gamma \vdash q \twoheadrightarrow q_1, \ldots, q_n \searrow \Gamma'}{\Gamma \vdash B \twoheadrightarrow \{q_1\}, \ldots, \{q_n\} \searrow \Gamma'}$$

$$\frac{\Gamma \vdash e_1 \twoheadrightarrow \vec{e_1} \searrow \Gamma' \qquad \Gamma' \vdash e_2 \twoheadrightarrow \vec{e_2} \searrow \Gamma''}{\Gamma \vdash (e_1, e_2) \twoheadrightarrow \bigcup_i \bigcup_k \{(e_1^i, e_2^k)\} \searrow \Gamma'}$$

Note 26. *Both pattern split and pair split are (clearly) highly non-deterministic. If implemented naively, neither will terminate when confronted with a fixed-point expression. Therefore, an actual implementation needs a good heuristic to determine when to stop splitting an expression. See* Section 5.6.1 *for one approach that works in many cases.*

$\boxed{B \searrow^{\{\}} p}$ **Comprehension forcing.** Comprehension forcing tries to create a pattern p out of a closed bound B. The attempt may fail; in this case, computation needs to backtrack.

$$\frac{\Gamma \vdash B \searrow^{!} c}{B \searrow^{\{\}} c} \qquad \frac{\Gamma \vdash B \searrow^{!} \mathsf{let}\ \rho\ \mathsf{in}\ \{p\} \qquad (\!|p|\!)_\rho \searrow^{\mathsf{P}} p'}{B \searrow^{\{\}} p'}$$

$$\frac{\Gamma \vdash B \searrow^{!} \mathsf{let}\ \rho\ \mathsf{in}\ (B_1, B_2)}{B \searrow^{\{\}} (x_1 <: \mathsf{let}\ \rho\ \mathsf{in}\ B_1,\ x_2 <: \mathsf{let}\ \rho\ \mathsf{in}\ B_2)}\ x_1, x_2 \text{ fresh}$$

$\boxed{\Gamma \vdash p <: B \longrightarrow_{x/q} p' \searrow \Gamma'}$ **Bound propagation.** Bound propagation replaces a variable x in a pattern p by its bound. In order to do this, it tries to force the bound into comprehension form.

The bound parameter B is used to propagate bound information inwards. It is not logically an input to the relation.

Logically, Γ, p, and x are inputs; q, p', and Γ' are outputs.

$$\frac{B \searrow^{\{\}} q}{\Gamma \vdash x <: B \longrightarrow_{x/q} q \searrow \Gamma} \qquad \frac{\Gamma \vdash p <: (B \mathbin{\overline{\cap}} A) \longrightarrow_{x/q} p' \searrow \Gamma'}{\Gamma \vdash (p <: A) <: B \longrightarrow_{x/q} (p' <: A) \searrow \Gamma'}$$

$$\frac{\Gamma \vdash B \prec B_1,\ B_2 \searrow \Gamma_1 \qquad \Gamma_1 \vdash p_1 <: B_1 \longrightarrow_{x/q} p_1' \searrow \Gamma_2 \qquad (\!|p_2|\!)_{x=\{q\}} \searrow^{\mathsf{P}} p_2'}{\Gamma \vdash (p_1, p_2) <: B \longrightarrow_{x/q} (p_1',\ p_2') \searrow \Gamma_2}$$

$$\frac{\Gamma \vdash B \prec B_1,\ B_2 \searrow \Gamma_1 \qquad \Gamma_1 \vdash p_2 <: B_2 \longrightarrow_{x/q} p_2' \searrow \Gamma_2}{\Gamma \vdash (p_1, p_2) <: B \longrightarrow_{x/q} (p_1,\ p_2') \searrow \Gamma_2}$$

Note 27. *Bound propagation rounds bounds upwards, possibly losing information.*

Note 28. *Bound propagation fails if it is unable to make any significant changes to the pattern p. In particular, this is true when trying to propagate a bound that contains very little information (or, equivalently, too many subcases).*

5 Implementation notes

5.1 Parsing

The parser is implemented in `Parser.hs`. It is an ordinary backtracking parser implemented using the Parsec library [LM12].

There are some differences between the syntax used in these notes and the one actually read by the parser:

- The arrows used in fixed-point and function definitions are rendered as their usual ASCII equivalent, `->`.
- Fusions do not use the $\oplus$ operator, which is also not present in ASCII. Instead, the vertical bar, `|`, is used instead, in analogy to a logical "or".
- There is additional syntax for top-level declarations read from files, introduced by the `val` keyword. Declarations support an optional bound annotation. In addition, they support specifying arguments as in our abbreviated let-syntax (Section 2) that are in scope both in the declared value and in the specified bound.
- By prefixing an expression or pattern with an exclamation mark (`!`), it becomes *strictness-annotated*. The meaning of a strictness annotation depends on the context. Strictness-annotated patterns, for instance, force weak-head reduction of the matched-against value during pattern matching. See Section 5.3 and Section 5.6.2 for more use cases for strictness annotations.
- There is an infix operator `|>` that provides a syntax for reverse application, i. e., `a |> b` is equivalent to `b a`. This is useful for matching an expression against a pattern. See Listing 5.5 for a use case.
- There is an abbreviated form of function type syntax. In an expression context, `a -> b` is defined as being equivalent to **`fun`** `{(x <: a) -> b}` where `x` is a newly generated variable.

5.2 Global variables

Global declarations are stored in *signatures*, which store both values and the corresponding user-supplied type bounds. See `Signature.hs` for the implementation. In addition to the `Sig` type itself, the module provides a monad transformer `SigT`, which provides an implementation of `State` semantics with a signature as the state value. A version restricted to reading, `SigReaderT`, is provided as well.

5.3 Evaluation

The evaluator provides two primary procedures:

`eval` is the function representing the evaluator itself. It implements the rules specified in Section 3. The implementation differs from the formal system insofar as it distinguishes between two "modes", one of which rounds according to the supplied bound context, the other refraining from doing so. This is useful for debugging the type checker, since the context can be checked for missing parameters. The formal rules we specified above make the distinction on the fly instead; calling the `eval` function in non-rounding mode being equivalent to calling upon the evaluation predicate with an empty context.

Backtracking is implemented by the explicit use of the `ListT` monad transformer [Gil12].

`evalToplevel` takes a list of top-level declarations and processes them in order, calling into the type checker for verification of user-supplied type bounds, or, in the absence of such, initiating well-formedness checks. In the process, `evalTopLevel` stores the processed declarations in the signature, and outputs the result of evaluating the right-hand-side of the declaration.

`evalTopLevel` supports strictness annotations on the declared variable (see Section 5.1). Whenever it encounters a strictness annotation, it will call into `eval` instead of storing the supplied right-hand side expression as-is. This makes it possible to have the top-level loop compute more meaningful output than would otherwise be produced.

5.4 Type checking

Type checking, too, is largely a direct translation of the formal system developed in this work. The implementation is monadic, based on a `TypeChecker` monad, which for convenience is an instance of `StateT` carrying along a state variable containing the following components:

`context` represents the bound context Γ, which stores the known bounds of parameters. Instead of explicitly passing and returning contexts Γ, the code implementing the type-checking rules modifies this state variable (see function `remember`).

`sig` contains the signature, which is not present in the formal system.

`idsupply` is a value of type `Data.Supply.Supply Integer`, which is provided by the `value-supply` library [Dia11]. It is used for the generation of unique parameters and variables.

The underlying monad is effectively an implementation of the free `MonadPlus` structure, supplied by the `TreeChecker` module. Its declaration is given in Listing 5.1.

Listing 5.1 The `SearchTree` and `Checker` data structures used by the type checker.

```
data SearchTree a
  = None
  | Result a
  | ChoiceNode (SearchTree a) (SearchTree a)
  | forall b. Step (b -> SearchTree a, SearchTree b)

data Checker err a = Checker (SearchTree (Either err a))
```

The `TreeChecker` module provides a number of implementations of computing the result of a `Checker` expression; the user can thus decide to use either depth-first (`runSearchDFS`) or breadth-first search (`runSearchBFS`), iterative deepening (`runSearchIterDFS`), or a parallel tree traversal based on the `unamb-custom` library [Pal08] (`runSearchUnamb`). The type checking module uses depth-first search by default.

The primary entry point to the type checker is the `check` function, which implements the basic type-checking rules described in Section 4.1. Since it is monadic, it must be called from external code by way of one of the type-checking trampolines, `runTypeChecker` and `runTypeCheckerND`:

`runTypeCheckerND` runs a function whose result lives in the `TypeChecker` monad, returning all possible results or, in the case of failure, a set of all error messages collected from all failed branches of the computation.

`runTypeChecker` is like `runTypeCheckerND`, but in the case of success, returns a single result only. This is usually sufficient, since the primary type-checking entry point, `check`, does not return a useful result in the case of success.

5.5 Additional features

In order to be practical, the code provides some features above and beyond what the formal system specifies.

5.5.1 Top-level definitions

Top-level definitions are introduced by the `val` keyword and may contain an optional bound annotation as in Listing 5.2.

Listing 5.2 Identity on the natural numbers in TypeCore.

```
val nat = 0 | ('s, nat)

val id_nat <: nat -> nat = fun
  { 0      -> 0
  ; ('s,m) -> ('s,m) }
```

Note the use of the abbreviated bound expression (`nat -> nat` $\equiv$ `fun {(n<:nat) -> nat}`) as described in Section 5.1.

As described above, the interpreter will process the top-level definitions given in an input file one by one, initiating type checks as appropriate. Since unannotated values are still checked for well-formedness, an ill-formed application such as the one depicted in Listing 5.3 will result in an error raised by the type checker.

Listing 5.3 An ill-formed application.

```
val g = 'g
val f = g 'arg
```

Type annotations are not usually required except in recursive function definitions where they are necessary to check recursive calls. The following file will type-check just fine, for example, since `id`, though initially stored as being of type $\top$, will be "unrounded" to itself (i. e., fun $\{x \to x\}$) during the application check after the failed attempt to do the check using its bound (Listing 5.4).

Listing 5.4 "Unrounding" of a function.

```
val id = fun {x -> x}
val zero = id 'zero
```

5.5.2 Integer arithmetic

In addition to symbols, the system features integers as another type of literal. A couple of built-in functions are provided:

`plus`, `minus`, `times`, `div` are the usual arithmetic primitives. They are curried, and declared to be of type $int \to int \to int$.

`pred`, `succ` are the predecessor and successor functions. Though technically redundant in the presence of `plus` and `minus`, they are sufficiently useful for the definition of recursive functions that they warrant their own built-ins.

`zerop` is an operator of type $int \to ('true \oplus 'false)$ that returns the symbol $'true$ if the supplied argument is the number 0, and the symbol $'false$ otherwise.

Note that there is no built-in boolean type, so $'true$ and $'false$ are regular literals. Instead of an `if` primitive, which is not supplied by our system, the $'true$ and $'false$ cases can be distinguished by way of pattern matching. See Listing 5.5 for an example. Note the use of the reverse-application syntax introduced in Section 5.1.

Listing 5.5 Factorial in TypeCore

```
val factorial (n <: int) <: int =
  zerop n |> fun { 'true  -> 1
                 ; 'false -> times n (factorial (pred n)) }
```

5.5.3 Read–eval–print loop

Module `Repl` implements a read–eval–print loop (or REPL) that can be accessed by running the `typecore` program with no arguments. It supports top-level definitions using the `val` syntax, including type-annotated definitions.

An example interaction is given in Listing 5.6 (slightly reformatted for improved readability and to account for minor deficiencies in the printer).

5.6 Conflict resolution

As noted before, certain predicates, notably type splitting (Section 4.12) and application checking (Section 4.2) contain ambiguous cases in which more than one rule is applicable. In the case of type splitting, the resulting non-algorithmicity appears particularly egregious, for while application checking can at least do meaningful backtracking across the two possible branches, type splitting as implemented naively inevitably recurs indefinitely.

5.6.1 Type and pattern split

As defined in our formalism, type and pattern splits are highly non-deterministic. In the general case, there is no good way of knowing when to stop unfolding an expression into ever more subcases. In the cases we actually need type splitting in, however, there is a very effective and straight-forward heuristic.

All cases we use type splitting in have in common that the split occurs so that matching against a pattern does not require destroying dependencies between components of a dependent pair. Therefore, there is an obvious stopping condition for the splitting process: Stop as soon as another split cannot possibly change the outcome of a pattern match. This means that we need to simply step through the expression to be split and the patterns to be matched against in tandem. We need to proceed splitting as long as the patterns contains a choice point, i. e., a literal, pair, or bound. As soon as the pattern cases we need to match are all trivial (i. e., variables), we can stop the splitting process.

A complication arises when we encounter a bound that itself is a pattern (in the form of a comprehension). In this case, since the pattern can include dependent pairs, we need to play it safe and integrate the pattern into the set of patterns we use to determine the stopping condition.

In addition, in case we split a pair pattern, we need to temporarily sever the link between the left and right components of the pattern (by generating bounded parameters and substituting them for the pattern variables occurring in the bounds) in order to be able to close the bounds of the right component; but immediately after the return of the recursive splitting call, we can undo this separation by reversing the original substitution. We might still lose some information this way, but in case the recursive call did not actually need to split the affected parameters, there is no information loss at all.

The algorithm is thus a relatively straight-forward recursion. See the definitions of `structuralPatternSplit` and `structuralSplit` in the `TypeChecker` module for the gritty details.

5.6.2 Compile-time versus run-time application

In Section 4.2, we noted that when confronted with an expression $f\,e$, the system needs to decide whether to do a limited form of compile-time type inference, or simply evaluate the expression.

Unfortunately, neither approach works all of the time. Clearly, evaluation is much too strong in general; unfolding even a very simple function call destructuring an argument of a recursive type like *nat* will immediately result in non-termination of the type checker.

Therefore, in the usual case, the limited form of inference described by the (*@check*) rule is the right choice.

In some cases, on the other hand, the (*@check*) rule is simply too weak. In particular, this is the case in the presence of "type" arguments, the reason being that the pattern matching rules have no way of distinguishing between an argument that is known to be *a subtype of* some parameter a, and an argument that is actually *equivalent to* a. Moreover, even if pattern matching were able to distinguish the two cases, it would not have a way of storing that information in the typing context Γ, which only stores bounds, not equivalences.

Because of this, we need some way of identifying the cases in which compile-time evaluation needs to happen. We have decided not to try to propose an appropriate heuristic in this work. Instead, our system takes a dual approach:

1. The type checker tries the (*@check*) rule by default. If this leads to backtracking, the (*@eval*) rule is tried instead.

2. In the cases where the default leads to non-termination, the system enables the user to annotate applications using strictness syntax (see Section 5.1). For esthetic reasons, we have decided to place the annotation on the argument (i. e., an annotated application is of the form $f\ !e$). This is an arbitrary choice. The annotation could, of course, also have been placed on the applicand ($!f\ e$) or around the application itself ($!(f\ e)$); but placing it on the argument has the advantage of not requiring any bracketing when more than one argument is supplied (e. g., a polymorphic *map* function, as present in most functional programming languages, can be called as `map !a !b some_function some_list`, where a and b are the input and output type parameters, respectively).

As far as we can tell, this approach works well in practice; at least, the number of annotations required is small enough not to have bothered us in writing our set of test cases.

Listing 5.6 Transcript of a REPL session.

```
TypeCore (C) LMU Munich
> 1
1

> plus 1 2
3

> plus 1 2 3
The following exception occurred:
   user error (["Attempted application of int type"])

> val if
   <: fun {(b <: ('true | 'false)) x y -> x | y}
    = fun {b x y -> b |> fun {'true -> x; 'false -> y}}
val if
 <: fun {(b <: ('true | 'false)) -> fun {x -> fun {y ->
      (x | y)}}}
  = fun {b -> fun {x -> fun {y ->
      fun {'true -> x; 'false -> y} b}}}

> val expt <: int -> int -> int  =
    fun {(x<:int) (n<:int) ->
      if (zerop n) 1 (times x (expt x (pred n)))}
val expt <: (int -> (int -> int)) =
  fun {(x <: int) -> fun {(n <: int) ->
    if (zerop n) 1 (times x (expt x (pred n)))}}

> expt 2 16
65536

> val pow2 <: int -> int = expt 2
val pow2 <: (int -> int) =
  let {x = 2} in
    fun {(n <: int) -> if (zerop n) 1 (times x (expt x (pred n)))}

> pow2 16
65536
```

6 Related work

6.1 Pure subtype systems

Pure subtype systems [Hut10] have been proposed as a way of unifying types and values by replacing type-checking with subtyping. In contrast to TypeCore, the proposed system in its basic form lacks type disjunctions and recursion. The system is explored more fully in [Hut09], which investigates subtype-based typing in the context of object class hierarchies.

6.2 Soft typing

Soft typing [CF91][WC94] is a generalization of Hindley–Milner type inference for dynamically typed programming languages such as Scheme. It supports type recursion and union types, which it is able to infer from the use of constructors similarly to TypeCore. One of the major innovations of soft typing, as exemplified by the Typed Racket language [THF10], is *occurrence typing*: dynamic type predicates (such as `pair?`) are used to refine known union types at compile-time. For example, an expression such as `(if (number? x) (cons x x) '())` will be inferred to be of type $null \cup (cons\ number\ number)$. Occurrence typing poses an interesting alternative to our pattern-matching-based approach in the presence of abstract types.

6.3 Recursive union and intersection types

F. Damm [Dam94] has proposed a representation of types based on sets of trees satisfying a regularity condition in order to ensure decidability of subtyping. The proposed family of type systems supports type recursion as well as union and intersection, but lacks polymorphic function types as well as any kind of dependent types.

6.4 Polymorphic variants

The OCaml language [INR11], based on a Hindley–Milner-style type system [GMM+78], sports a feature called polymorphic variants [Gar98][Gar04], which permit the use of data constructor tags without prior declaration. To this effect, OCaml provides a kind of "ad-hoc" type inference for variants quite similar to the handling of pairs and literals in TypeCore. Just as in TypeCore, a `not` function can be defined by using undeclared tags (see Listing 6.2). Variant types can be subtyped (see Listing 6.3). Type recursion is also supported, including type inference, enabling the use of polymorphic variants to build inductive data types (Listing 6.4). In contrast to TypeCore, however, polymorphic variants are not dependently typed and do not permit the use of arbitrary values as variant cases.

Listing 6.1 OCaml `not` function, based on polymorphic variants.

```
# let not = function ‘True -> ‘False
                   | ‘False -> ‘True;;
val not : [< ‘False | ‘True ] -> [> ‘False | ‘True ]
```

Listing 6.2 OCaml `not` function, based on polymorphic variants.

```
# let not = function ‘True -> ‘False
                   | ‘False -> ‘True;;
val not : [< ‘False | ‘True ] -> [> ‘False | ‘True ]
```

Listing 6.3 OCaml `not` function, restricted to a `‘False` argument by subtyping.

```
# let notFalse : [< ‘False] -> [> ‘False | ‘True] = not;;
val notFalse : [< ‘False ] -> [> ‘False | ‘True ]
```

Listing 6.4 Recursive variants in OCaml.

```
# let rec id_nat =
    function ‘zero   -> ‘zero
           | ‘succ x -> ‘succ (id_nat x);;
val id_nat :
  ([< ‘succ of ’a | ‘zero ] as ’a) ->
  ([> ‘succ of ’b | ‘zero ] as ’b)
```

7 Conclusion and outlook

As our calculus and implementation demonstrate, a type system based on the mereological parthood relation rather than set-theoretic elementhood is both feasible and can be made easy to understand and use by programmers.

The system raises interesting questions regarding type systems in general, as well as their connection to partial evaluation (see Section 4.2) and relational calculi. In particular, it might be interesting to investigate a statically typed relational calculus using a mereological type system; indeed, any logic-based or database system that wants to integrate a functional language is a good fit for mereological typing.

All that said, the system does have some shortcomings and unsolved problems.

Listing 7.1 Predicate logic in TypeCore.

```
val T
  <: fun {(n <: nat) → 'trivial}
  = fun { 0                 → 'trivial
        ; ('s, (x <: nat)) → T x      }

val allT
  <: fun {(n <: nat) → T n}
  = fun { 0                 → 'trivial
        ; ('s, (x <: nat)) → allT x   }
```

Lack of an algebraic simplifier. The system as of now lacks an algebraic simplifier, which is a standard feature of dependently-typed programming languages. This makes it impossible to use the system as a theorem prover, among other things. In particular, the system cannot type-check even a simple program emulating predicate logic such as the one given in listing 7.1.

Fortunately, though out of the scope of this work, we do not see a major obstacle to implementing an algebraic simplifier for TypeCore.

Lack of a good application-check heuristic. As discussed in Section 4.2, the system needs to non-deterministically decide whether to unfold an application or recur during a type check. It is not clear how the system should be able to tell, without human intervention, which of the two paths it ought to take.

In fact, this may or may not be a computationally soluble problem. Existing systems solve it by separating strictly between types and run-time expressions; but this would be an obvious detractor in the case of a system like TypeCore, whose most significant distinguishing factor is the absence of such separation.

For now, explicit annotations forcing compile-time unfolding seem to be an adequate workaround. In the future, heuristics might be discovered that significanlty lift the (in our experience, already rather light) burden from the programmer as far as feasible.

We are confident that future efforts can overcome these shortcomings, resulting in an elegant and reasonably complete system able to pose an alternative to existing dependent type systems.

References

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of AGDA - a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer-Verlag, 2009.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 278–292. ACM, 1991.

[CV99] Roberto Casati and Achille C. Varzi. *Parts and Places: The Structures of Spatial Representation*. MIT Press, Cambridge, MA, 1999.

[Dam94] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In Hagiya and Mitchell [HM94], pages 687–706.

[Dia11] Iavor S. Diatchki. value-supply-0.6. http://hackage.haskell.org/package/value-supply, 2011.

[Gar98] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.

[Gar04] Jacques Garrigue. Typing deep pattern-matching in presence of polymorphic variants. In *JSSST Workshop on Programming and Programming Languages, Gamagori, Japan*, March 2004.

[Gil12] Andy Gill. mtl-2.1.2. http://hackage.haskell.org/package/mtl, 2012.

[GMM$^+$78] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 119–130. ACM Press, 1978.

[HM94] Masami Hagiya and John C. Mitchell, editors. volume 789 of *Lecture Notes in Computer Science*. Springer, 1994.

[Hut09] DeLesley. S. Hutchins. *Pure subtype systems: A type theory for extensible software*. PhD thesis, University of Edinburgh, 2009.

[Hut10] DeLesley S. Hutchins. Pure subtype systems. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 287–298. ACM, 2010.

[INR11] INRIA. The OCaml system. `http://caml.inria.fr/ocaml/index.en.html`, 2011.

[Jec11] Thomas Jech. Set theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, Center for the Study of Language and Information, Stanford University, Stanford, CA 94305-4115, Winter 2011 edition, 2011.

[LM12] Daan Leijen and Paolo Martini. parsec-3.1.3. `http://hackage.haskell.org/package/parsec`, 2012.

[Pal08] Luke Palmer. unamb-custom-0.13. `http://hackage.haskell.org/package/unamb-custom`, 2008.

[PM84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[SS94] Leon Shapiro and Ehud Y. Sterling. *The Art of* PROLOG*: Advanced Programming Techniques*. The MIT Press, April 1994.

[Ste98] Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Technische Fakultät, Universität Erlangen, 1998.

[THF10] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’10, pages 117–128, New York, NY, USA, 2010. ACM.

[WC94] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In LISP *and Functional Programming*, pages 250–262, 1994.

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 24.11.2012. ....