# Lecture Notes on Arithmetic Refinements

15-814: Types and Programming Languages
Frank Pfenning

Lecture 25
Thursday, December 3, 2020

## 1 Introduction

We have seen a number of benefits of types as an organizing principle in the design of programming languages. When statically checked, they provide the guarantees of *type safety* (summarized in the properties *preservation* and *progress*) and characterize the results of evaluation (via the *canonical forms theorem*). Static typing breaks down the properties of a whole program into individually checkable properties of the expressions, functions, and modules making up this program. They also allow us to express data abstraction and representation independence in a way that it can be enforced by a type-checker rather than just by convention.

So far, however, the properties that can be expressed in a type are rather rudimentary when compared to complete specifications. For example, any unary function on natural numbers has the type $nat \to nat$, including varieties such as the successor, predecessor, power of two, integer logarithm, etc.

In this lecture we explore how type theory supports the expression of a whole range of program properties, from *simple*, *recursive*, and *polymorphic* types we have seen so far to full "functional" program specifications. In the next lecture we will see how to go even further and capture *intensional* properties of programs, such as their computational complexity, entirely within the type.

A key idea is that of a *dependent type*, that is, a type that may *depend* on a value. General dependent type theory is highly expressive but also highly complex. It necessarily blends programming with theorem proving, which should not come as a surprise given our exploration of *Types as*

*Propositions* in Lecture 15. Even an introductory treatment would require a whole semester's worth of lectures. Despite this complexity, dependent type theories and related programming languages are becoming more widespread, including, for example Nuprl [C+86], Coq [BC04], Agda [Nor07], and Idris [Bra13].

Instead of tackling dependent type theories in their full generality, we explore *dependent refinement types* [XP99] that avoid the need for general theorem proving by restricting dependencies so that type-checking can be accomplished by a decision procedure. In this way we preserve the character of a programming language rather than asking the programmer to also prove their program correct. This comes at a cost: there are many program properties of interest that we will not be able to express.

## 2 Arithmetic Refinements

The particular instance of dependent refinement types we consider here are *arithmetic refinements*, that is, types may be *indexed* by arithmetic expressions. The exact language of arithmetic expressions is somewhat open-ended. For example, we may want to preserve decidability and restrict ourselves to *Presburger arithmetic* which has constants, addition, multiplication by a constant, and the propositions including conjunction, disjunction, negation, and quantification. Or we could allow more general expressions and "do our best" with heuristics for proving arithmetic expressions.

Instead of using integers $\mathbb{Z}$ we will use natural numbers $\mathbb{N}$, which is the same as $\mathbb{Z}$ where every variable $x$ is constrained by $x \geq 0$. Here are some examples of arithmetically indexed types we can express:

| | |
|---|---|
| *list* $\alpha$ $n$ | Lists of length $n$ |
| *bin* $n$ | Binary numbers of value $n$ |
| *nat* $n$ | Unary numbers of value $n$ |
| *stack* $\alpha$ $n$ | Stack with $n$ elements |
| *incstream* $n$ | Increasing streams of natural numbers $\geq n$ |
| *tree* $l$ $u$ | Trees of natural numbers $x$ with $l < x < u$ |

In each of these cases, there are two challenges:

1. expressing the type itself so it accurately captures the property we care about, and
2. giving correct types to the functions that operate on elements of the type.

That's over and above the general challenge to tie together the language, the type checker, and the decision procedure (or heuristic algorithm) for deciding the validity of propositions in arithmetic.

## 3 Example: Lists Indexed by Length

We would like *list* $\alpha$ $n$ to be a type. So *list* is now a function that maps types $\alpha$ and natural numbers $n$ to types. One can formally describe this via so-called *kinds* for *type constructors* or *type families* as in the system $F^\omega$ [Gir71] or the Calculus of Constructions [CH88]. We avoid a full formalization of this since we would like to focus on programming aspects of indexed types. Let's start with a type

$$\mathit{list}\ \alpha \cong (\mathbf{nil} : 1) + (\mathbf{cons} : \alpha \times \mathit{list}\ \alpha)$$

Let's first consider the **cons** branch. If the whole list has length $n$, then the tail of the list has length $n - 1$:

$$\mathit{list}\ \alpha\ n \cong (\mathbf{nil} : 1) + (\mathbf{cons} : \alpha \times \mathit{list}\ \alpha\ (n - 1))$$

We use the color blue to highlight all arithmetic expressions and propositions that belong to the refinement layer. The above is not quite correct because a list of length $0$ may have a tail of length $-1$, so we need to enforce the $n - 1 \geq 0$ in case that list is nonempty. We express this with a type in the form of $\phi \wedge \tau$ where $\phi$ is a proposition from arithmetic. We often refer to $\phi$ as a *constraint*.

$$\mathit{list}\ \alpha\ n \cong (\mathbf{nil} : 1) + (\mathbf{cons} : n > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (n - 1))$$

Finally, in the alternative **nil** we must constrain $n$ to be $0$, since nil always represents a list of length zero.

$$\mathit{list}\ \alpha\ n \cong (\mathbf{nil} : n = 0 \wedge 1) + (\mathbf{cons} : n > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (n - 1))$$

Before we more rigorously express how do handle arithmetic propositions and types such a $\phi \wedge \tau$, let's examine how they will be used. Let's start with the *constructor functions nil* and *cons*. We would like to have

$$\begin{array}{l} \mathit{nil} : \forall\alpha.\, \mathit{list}\ \alpha\ 0 \\ \mathit{nil} = \Lambda\alpha.\, \mathsf{fold}\ (\mathbf{nil} \cdot \langle\rangle) \end{array}$$

because *nil* constructs a list of length 0. Let's walk through the typing derivation of this.

$$\dfrac{\dfrac{\dfrac{\vdots}{\alpha\ \mathit{type} \vdash (\mathbf{nil} \cdot \langle\rangle) : (\mathbf{nil} : 0 = 0 \wedge 1) + (\mathbf{cons} : 0 > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (0-1))}\ \mathsf{tp/fold}}{\dfrac{\alpha\ \mathit{type} \vdash \mathsf{fold}\ (\mathbf{nil} \cdot \langle\rangle) : \mathit{list}\ \alpha\ 0}{\cdot \vdash \Lambda\alpha.\, \mathsf{fold}\ (\mathbf{nil} \cdot \langle\rangle) : \forall\alpha.\, \mathit{list}\ \alpha\ 0}\ \mathsf{tp/tplam}}$$

Note here that at the step where we unfolded the type we substituted $0$ for $n$, because the second index to *list* is 0. Using the rule for sums, we find

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\alpha\ \mathit{type} \vdash \langle\rangle : 0 = 0 \wedge 1}}{\alpha\ \mathit{type} \vdash (\mathbf{nil} \cdot \langle\rangle) : (\mathbf{nil} : 0 = 0 \wedge 1) + (\mathbf{cons} : 0 > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (0-1))}\ \mathsf{tp/tag}}{\dfrac{\alpha\ \mathit{type} \vdash \mathsf{fold}\ (\mathbf{nil} \cdot \langle\rangle) : \mathit{list}\ \alpha\ 0}{\cdot \vdash \Lambda\alpha.\, \mathsf{fold}\ (\mathbf{nil} \cdot \langle\rangle) : \forall\alpha.\, \mathit{list}\ \alpha\ 0}\ \mathsf{tp/tplam}}\ \mathsf{tp/fold}$$

At this point we have to show that $0 = 0$, in general employing assumptions from the context (although there are none here). This is a new judgment from arithmetic, so we write it as $\Gamma \models \phi\ \mathit{true}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\cdot \models 0 = 0\ \mathit{true}} \quad \dfrac{}{\alpha\ \mathit{type} \vdash \langle\rangle : 1}\ \mathsf{tp/unit}}{\alpha\ \mathit{type} \vdash \langle\rangle : 0 = 0 \wedge 1}\ \mathsf{tp/and}}{\alpha\ \mathit{type} \vdash (\mathbf{nil} \cdot \langle\rangle) : (\mathbf{nil} : 0 = 0 \wedge 1) + (\mathbf{cons} : 0 > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (0-1))}\ \mathsf{tp/tag}}{\dfrac{\alpha\ \mathit{type} \vdash \mathsf{fold}\ (\mathbf{nil} \cdot \langle\rangle) : \mathit{list}\ \alpha\ 0}{\cdot \vdash \Lambda\alpha.\, \mathsf{fold}\ (\mathbf{nil} \cdot \langle\rangle) : \forall\alpha.\, \mathit{list}\ \alpha\ 0}\ \mathsf{tp/tplam}}\ \mathsf{tp/fold}$$

Fortunately, $0 = 0$ is true, so this typing should be valid.

We observe here a new phenomenon, namely a typing rule that does not change the expression. It is in part this property which makes this system a system of *type refinement* rather than a full dependent type theory. In the general form:

$$\dfrac{\Gamma \models \phi\ \mathit{true} \quad \Gamma \vdash e : \tau}{\Gamma \vdash e : \phi \wedge \tau}\ \mathsf{tp/and/i}$$

Note that not all the assumptions in $\Gamma$ can actually be relevant to the truth of $\phi$, but we will gloss over this detail here.

Next, there should be counterpart, tp/and/e. Our form of conjunction should be *positive* (as we will see shortly), so the elimination should in the form of a case-like rule. However, this is difficult because we do not want the expression $e$ to change, but a case rule requires two expressions. Instead, we build the elimination rule into pattern matching. Recall from Lecture 12:

$$\begin{array}{lcll} \text{Expressions} & e & ::= & \ldots \mid \mathsf{case}\ e\ (bs) \\ \text{Patterns} & p & ::= & x \mid \langle p_1, p_2 \rangle \mid \langle\,\rangle \mid i \cdot p \mid \mathsf{fold}\ p \\ \text{Branches} & bs & ::= & \cdot \mid (p \Rightarrow e \mid bs) \end{array}$$

There were two relevant judgments beyond typing of expressions:

**Matching:** $\Gamma \vdash \tau \triangleright bs : \sigma$ which expresses a subject of type $\tau$ matches the branches $bs$, all of which have type $\sigma$.

**Patterns:** $\Gamma \Vdash p : \tau$ which expresses that pattern $p$ has type $\tau$.

This latter judgment was an early example of a *linear* judgment, because we wanted every variable in $\Gamma$ to occur exactly once in $p$.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \triangleright bs : \sigma}{\Gamma \vdash \mathsf{case}\ e\ (bs) : \sigma}\ \mathsf{case}$$

---

$$\frac{\Gamma' \Vdash p : \tau \quad \Gamma, \Gamma' \vdash e : \sigma \quad \Gamma \vdash \tau \triangleright bs : \sigma}{\Gamma \vdash \tau \triangleright (p \Rightarrow e \mid bs) : \sigma}\ \mathsf{tp/bs/alt} \qquad \frac{}{\Gamma \vdash \tau \triangleright (\cdot) : \sigma}\ \mathsf{tp/bs/none}$$

---

$$\frac{}{x : \tau \Vdash x : \tau}\ \mathsf{pat/var} \qquad \frac{\Gamma_1 \Vdash p_1 : \tau_1 \quad \Gamma_2 \Vdash p_2 : \tau_2}{\Gamma_1, \Gamma_2 \Vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2}\ \mathsf{pat/pair} \qquad \frac{}{\cdot \Vdash \langle\,\rangle : 1}\ \mathsf{pat/unit}$$

$$\frac{(k \in I) \quad \Gamma \Vdash p : \tau_k}{\Gamma \Vdash k \cdot p : \sum_{i \in I} (i : \tau_i)}\ \mathsf{pat/inject} \qquad \frac{\Gamma \Vdash p : [\rho\alpha.\,\tau/\alpha]\tau}{\Gamma \Vdash \mathsf{fold}\ p : \rho\alpha.\,\tau}\ \mathsf{pat/fold}$$

We now add to the pattern typing the rule

$$\frac{\Gamma, \phi\ \mathit{true} \Vdash p : \tau}{\Gamma \Vdash p : \phi \wedge \tau}\ \mathsf{pat/and}$$

In analogy with the typing rule for expressions, the pattern $p$ does not change in this rule.

Now let's return to the constructor and see if we can type

$\mathit{cons} : \forall\alpha.\, \forall n.\, \alpha \to \mathit{list}\ \alpha\ n \to \mathit{list}\ \alpha\ (n+1)$

Already we note that we need a universal quantifier over index expressions. It's not difficult to image what the rules for the universal quantifier might look like:

$$\frac{\Gamma, n : \mathbb{N} \vdash e : \tau}{\Gamma \vdash \lambda n.\, e : \forall x.\, \tau}\ \mathsf{tp/nlam} \qquad \frac{\Gamma \vdash e : \forall n.\, \tau \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash e\ t : [t/x]\tau}\ \mathsf{tp/napp}$$

$$\frac{}{\lambda n.\, e\ \mathit{value}}\ \mathsf{val/nlam} \qquad \frac{e \mapsto e'}{e\ t \mapsto e'\ t}\ \mathsf{step/napp}_1 \qquad \frac{}{(\lambda n.\, e)\ t \mapsto [t/n]\, e}\ \mathsf{step/napp/nlam}$$

Here, $t$ ranges over arithmetic terms. This is in effect the extension needed if we wanted to generalize the Curry-Howard interpretation of intuitionistic logic to include universal quantification, except that here we committed the quantification to be over natural numbers. We do not evaluate arithmetic terms because we think of them as serving the purpose of refined types, not additional computations. In fact, it is also possible to remove abstraction over and application to arithmetic terms from the language of expressions, but this complicates type-checking significantly.

Returning to the *cons* constructor for lists, here is what are hoping for.

$\mathit{cons} : \forall\alpha.\, \forall n.\, \alpha \to \mathit{list}\ \alpha\ n \to \mathit{list}\ \alpha\ (n+1)$
$\mathit{cons} = \Lambda\alpha.\, \lambda n.\, \lambda x.\, \lambda l.\, \mathsf{fold}\ (\mathbf{cons} \cdot \langle x, l\rangle)$

After discharging the abstractions into the context we arrive at the judgment

$$\alpha\ \mathit{type}, n : \mathbb{N}, x : \alpha, l : \mathit{list}\ \alpha\ n \vdash \mathsf{fold}\ (\mathbf{cons} \cdot \langle x, l\rangle) : \mathit{list}\ \alpha\ (n+1)$$

Unfolding the recursive type, this holds if we can show:

$\alpha\ \mathit{type}, n : \mathbb{N}, x : \alpha, l : \mathit{list}\ \alpha\ n \vdash$
$(\mathbf{cons} \cdot \langle x, l\rangle) : (\mathbf{nil} : \ldots) + (\mathbf{cons} : n+1 > 0 \wedge \alpha \times \mathit{list}\ \alpha\ ((n+1)-1))$

Selecting the **cons** alternative of the sum and noting that $n : \mathbb{N} \models n+1 > 0$ we arrive at

$$\alpha\ \mathit{type}, n : \mathbb{N}, x : \alpha, l : \mathit{list}\ \alpha\ n \vdash l : \mathit{list}\ \alpha\ ((n+1)-1)$$

Clearly, this should hold because for every $n : \mathbb{N}$ we have that $n = (n+1)-1$ and therefore also $\mathit{list}\ \alpha\ n = \mathit{list}\ \alpha\ ((n+1)-1)$. Here, we think of equality as meaning the two types are inhabited by exactly the same values (for every $\alpha$ and $n$).

To handle this rigorously, we should allow a rule of type conversion and some rules to formalize some notion of type equality (see Mini-Project 1.1 or [LBN17]). We should have at least the rule of *type conversion* and allow provably equal index terms to be used interchangeably.

$$\frac{\Gamma \vdash \tau = \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau}\ \mathsf{tp/conv} \qquad \frac{\Gamma \vdash \tau = \tau' \quad \Gamma \models t = t'}{\Gamma \vdash \tau\ t = \tau'\ t'}\ \mathsf{conv/idx}$$

Different notions of type equality or subtyping are a vast subject, so we just pragmatically assume we have at least the two rules above to allow judgments such as the typing of *cons* and other examples.

## 4 Checking Recursive Functions

As a generic example of a recursive function, we consider typing the append function for lists. We repeat the indexed type of lists for reference and the expected type for *append*.

$$\mathit{list}\ \alpha\ n \cong (\mathbf{nil} : n = 0 \wedge 1) + (\mathbf{cons} : n > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (n-1))$$

$$\mathit{append} : \forall\alpha.\, \forall n.\, \forall k.\, \mathit{list}\ \alpha\ n \to \mathit{list}\ \alpha\ k \to \mathit{list}\ \alpha\ (n+k)$$

We first write the definition, which is straightforwardly extended from the usual (unindexed) definition.

$$\begin{array}{l}\mathit{append} = \mathsf{rec}\ \mathit{append}.\, \Lambda\alpha.\, \lambda n.\, \lambda k.\, \lambda l_1.\, \lambda l_2. \\ \qquad \mathsf{case}\ l_1\ (\ \mathsf{fold}\ (\mathbf{nil} \cdot \langle\,\rangle) \Rightarrow l_2 \\ \qquad\qquad \mid \mathsf{fold}\ (\mathbf{cons} \cdot \langle x, l_1' \rangle) \Rightarrow \mathit{cons}\ \alpha\ ((n-1)+k)\ x\ (\mathit{append}\ \alpha\ (n-1)\ k\ l_1'\ l_2)\ )\end{array}$$

Here, the recursive call to append is at $n-1$ and $k$ because $l_1' : \mathit{list}\ \alpha\ (n-1)$. This means the recursive call

$$\mathit{append}\ \alpha\ (n-1)\ k\ l_1'\ l_2 : \mathit{list}\ \alpha\ ((n-1)+k)$$

so the index argument to *cons* should be $(n-1)+k$. However, for this to be well-formed we need to know $n > 0$. Because $l_1 : \mathit{list}\ \alpha\ n$, when we match $l_1$ against fold $(\mathbf{cons} \cdot \langle x, l' \rangle)$ we find that

$$n > 0\ \mathit{true}, x : \alpha, l' : \mathit{list}\ \alpha\ (n-1) \Vdash \mathsf{fold}\ (\mathbf{cons} \cdot \langle x, l' \rangle) : \mathit{list}\ \alpha\ n$$

and we obtain the necessary assumption $n > 0$.

The result of applying the *cons* constructor will have type $((n-1)+k)+1$ and we have

$$n : \mathbb{N}, k : \mathbb{N}, n > 0 \; \mathit{true} \models ((n-1)+k)+1 = n+k \; \mathit{true}$$

where $n + k$ is the required result type.

It remains to check the case of **nil**. In that case, pattern matching $l_1$ against *fold* $(\mathbf{nil} \cdot \langle\rangle)$ obtains the information that $n = 0$. Then $l_2 : \mathit{list}\ \alpha\ k$ and $n : \mathbb{N}, k : \mathbb{N}, n = 0 \; \mathit{true} \models k = 0 + k \; \mathit{true}$ gives us the type-correctness of the first branch.

It is a straightforward variation on this theme to check, for example, that *reverse* preserves the length of the list.

## 5 Contradictory Constraints

Consider the type $\mathit{list}\ \alpha\ 1$. It describes a list with just one element. Consequently, the following case statement should be cover all possible cases:

$\mathit{the} : \forall\alpha.\, \mathit{list}\ \alpha\ 1 \to \alpha$
$\mathit{the} = \Lambda\alpha.\, \lambda l.\, \mathsf{case}\ l\ (\mathsf{fold}\ (\mathbf{cons} \cdot \langle x, \mathsf{fold}\ (\mathbf{nil} \cdot \langle\rangle)\rangle) \Rightarrow x)$

For the sake of argument, say we added a nil branch:

$\mathit{the} : \forall\alpha.\, \mathit{list}\ \alpha\ 1 \to \alpha$
$\mathit{the} = \Lambda\alpha.\, \lambda l.\, \mathsf{case}\ l\ (\ \mathsf{fold}\ (\mathbf{cons} \cdot \langle x, \mathsf{fold}\ (\mathbf{nil} \cdot \langle\rangle)\rangle) \Rightarrow x$
$\qquad\qquad\qquad\qquad \mid \mathsf{fold}\ (\mathbf{nil} \cdot \langle\rangle) \Rightarrow e\ )$

Playing through the rules for pattern matching we note that when we reach $e$ it will be checked with

$$\alpha\ \mathit{type}, l : \mathit{list}\ \alpha\ 1, 0 = 1\ \mathit{true} \vdash e : \alpha$$

Of course, we have no way to return an element of the parameter type $\alpha$. But we shouldn't have to since we are in an impossible branch! There is no value $v$ such that $v : \mathit{list}\ \alpha\ 1$ and $v = \mathsf{fold}\ (\mathbf{nil}\ \langle\rangle)$. So type-checking this branch should succeed because it is impossible for it ever to be taken. We therefore have the rule

$$\frac{\Gamma \models \bot\ \mathit{true}}{\Gamma \vdash e : \tau}\ \mathsf{tp/contra}$$

By such reasoning, when the exhaustiveness of pattern matching is checked then the absence of impossible branches should not be flagged, just like in the first definition of *the*.

## 6 Binary Numbers and Singleton Types

Numbers in binary representation present some new challenges. We would like *bin* $n$ be the type of binary numbers with value $n$. Note that the binary numbers (or unary numbers) as recursive types are different from the natural numbers $\mathbb{N}$ used in the index domain. In a fully dependent type system, this does not have to be so, but it is more difficult to obtain the power of the decision procedures for Presburger arithmetic.

Here is the first attempt

$$\begin{array}{rl} \mathit{bin}\ n \cong & (\mathbf{e} : n = 0 \wedge 1) \\ & + (\mathbf{b0} : \mathsf{even}\ n \wedge \mathit{bin}\ (n/2)) \\ & + (\mathbf{b1} : \mathsf{odd}\ n \wedge \mathit{bin}\ ((n-1)/2)) \end{array}$$

But even though $\mathsf{even}\ n \triangleq \exists k.\, 2k = n$ can be defined in Presburger arithmetic, integer division can not (directly). So we introduce an existential quantifier to span the condition and the recursive occurrence of the type.

$$\begin{array}{rl} \mathit{bin}\ n \cong & (\mathbf{e} : n = 0 \wedge 1) \\ & + (\mathbf{b0} : \exists k.\, n = 2k \wedge \mathit{bin}\ k) \\ & + (\mathbf{b1} : \exists k.\, n = 2k + 1 \wedge \mathit{bin}\ k) \end{array}$$

If we would like to prevent leading zeros in the representation we can constrain $k$ (or $n$) in the case of a bit 0.

$$\begin{array}{rl} \mathit{bin}\ n \cong & (\mathbf{e} : n = 0 \wedge 1) \\ & + (\mathbf{b0} : \exists k.\, k > 0 \wedge n = 2k \wedge \mathit{bin}\ k) \\ & + (\mathbf{b1} : \exists k.\, n = 2k + 1 \wedge \mathit{bin}\ k) \end{array}$$

With the particular restriction we see two phenomena that didn't exist with lists:

1. There are values $v : \mathit{bin}$ such that there is no index $n$ such that $v : \mathit{bin}\ n$. In other words, some values (namely those with leading zeros) are no longer well-typed, sharpening the canonical form theorem.

2. Each type *bin* $n$ is inhabited by exactly one value, namely *the* binary representation of $n$. We call such a type a *singleton type*, characterizing its value precisely.

We do not write any programs over refined binary numbers; see Exercise 1, but we can state the types of the constructors:

$e : \mathit{bin}\ 0$
$\mathit{b0} : \forall n.\, n > 0 \supset \mathit{bin}\ n \to \mathit{bin}\ (2n)$
$\mathit{b1} : \forall n.\, \mathit{bin}\ n \to \mathit{bin}\ (2n+1)$

We see that the type of *b0* requires a constraint implication. Perhaps that is not so surprising since we have a $\exists/\forall$ duality but also related $\wedge/\supset$ duality. We complete our language in the next section to include the additional quantifier.

## 7 Completing the Language

We summarize the language and supply the rules we have omitted so far. As mentioned before, the language of arithmetic terms and propositions is somewhat open-ended but with certain extensions we will lose decidability.

| | | | |
|---|---|---|---|
| Types | $\tau$ | $::=$ | $\ldots \mid \phi \wedge \tau \mid \phi \supset \tau \mid \exists n.\, \tau \mid \forall n.\, \tau$ |
| Expressions | $e$ | $::=$ | $\ldots \mid \langle t, e\rangle \mid \lambda n.\, e \mid e\ t$ |
| Patterns | $p$ | $::=$ | $\ldots \mid \langle n, p\rangle$ |
| Constants | $c$ | $::=$ | $0 \mid 1 \mid \ldots$ |
| Arith. Terms | $t$ | $::=$ | $c \mid t_1 + t_2 \mid t_1 - t_2 \mid c\,t \mid \ldots$ |
| Arith. Props | $\phi$ | $::=$ | $t_1 = t_2 \mid t_1 > t_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \supset \phi_2 \mid \top \mid \bot \mid \exists n.\, \phi \mid \forall n.\, \phi$ |
| Contexts | $\Gamma$ | $::=$ | $\ldots \mid \Gamma, n : \mathbb{N} \mid \Gamma, \phi\ \mathit{true}$ |

Besides the arithmetic entailment $\Gamma \models \phi\ \mathit{true}$ we also use the judgment $\Gamma \vdash t : \mathbb{N}$, which checks that $t$ is well-formed and that $\Gamma \models t \geq 0\ \mathit{true}$.

$$\frac{\Gamma \models \phi\ \mathit{true} \quad \Gamma \vdash e : \tau}{\Gamma \vdash e : \phi \wedge \tau}\ \mathsf{tp/and/i} \qquad \frac{\Gamma, \phi\ \mathit{true} \Vdash p : \tau}{\Gamma \Vdash p : \phi \wedge \tau}\ \mathsf{pat/and}$$

$$\frac{\Gamma, \phi\ \mathit{true} \vdash e : \tau}{\Gamma \vdash e : \phi \supset \tau}\ \mathsf{tp/imp/i} \qquad \frac{\Gamma \vdash e : \phi \supset \tau \quad \Gamma \models \phi}{\Gamma \vdash e : \tau}\ \mathsf{tp/imp/e}$$

$$\frac{\Gamma \vdash t : \mathbb{N} \quad \Gamma \vdash e : [t/n]\,\tau}{\Gamma \vdash \langle t, e\rangle : \exists n.\, \tau}\ \mathsf{exists/i} \qquad \frac{\Gamma, n : \mathbb{N} \Vdash p : \tau}{\Gamma \Vdash \langle n, p\rangle : \exists n.\tau}\ \mathsf{pat/exists}$$

$$\frac{\Gamma, n : \mathbb{N} \vdash e : \tau}{\Gamma \vdash \lambda n.\, e : \forall n.\, \tau}\ \mathsf{forall/i} \qquad \frac{\Gamma \vdash e : \forall n.\, \tau \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash e\,t : [t/n]\tau}\ \mathsf{forall/e}$$

$$\frac{\Gamma \models \bot\ \mathit{true}}{\Gamma \vdash e : \tau}\ \mathsf{tp/contra}$$

## 8 Some Additional Types

In this section we represent the encoding of some additional types using indexed refinement.

$$\textit{incstream}\ n \cong \exists k.\, k \geq n \wedge (\mathbf{hd} : \textit{bin}\ k)\ \&\ (\mathbf{tl} : \textit{incstream}\ k)$$
$$\textit{stack}\ \alpha\ n \cong (\mathbf{push} : \alpha \to \textit{stack}\ \alpha\ (n+1)) \\ \&\ (\mathbf{pop} : (\mathbf{none} : n = 0 \wedge 1) + (\mathbf{some} : n > 0 \wedge \alpha \times \textit{stack}\ \alpha\ (n-1)))$$
$$\textit{tree}\ l\ u \cong (\mathbf{leaf} : 1) + (\mathbf{node} : \exists n.\, l < n \wedge n < u \wedge \textit{tree}\ l\ n \times \textit{bin}\ n \times \textit{tree}\ n\ u)$$

## Exercises

**Exercise 1** Using the example for natural numbers in binary form, explore the following functions. Highlight in each case the constraints that would have to be checked to verify type correctness. Which of these are checkable in Presburger arithmetic?

(i) Provide the type and write the implementation of the successor function.

(ii) Provide the type and write the implementation of the predecessor function on positive numbers.

(iii) Provide the type and write the implementation of the addition functions.

(iv) Provide the type and write the implementation of the exponential function specified mathematically with $\mathsf{exp2}(x) = 2^x$.

**Exercise 2** Give a definition for natural numbers in unary form so that every type *nat* $n$ is the singleton with the representation of $n$ in unary form. Then revisit the functions in Exercise 1.

**Exercise 3** Write an implementation of stacks as specified in Section 8, perhaps recycling an earlier implementation, explicit stating the indexed types for every function you need. Isolate the constraints that have to be checked in each case and verify that they are true.

**Exercise 4** Specify the type of queues with $n$ elements and revisit your implementation of queues with two stacks. Does it type-check? You may assume that the stack operations check according to the type in Section 8 and Exercise 3.

**Exercise 5** Consider the *ordered trees of natural numbers* as specified in Section 8. These trees can be seen as an efficient representation of sets of natural numbers, assuming they are sufficiently balanced (a requirement we ignore in this exercise).

Attempt each of the following steps with the goal of completing them all to arrive at an implementation where the ordering invariant of binary search trees is enforced via type-checking. The ordering invariant states that for a node with element $n$, all elements in the left subtree are strictly less than $n$ and all elements in the right subtree are strictly greater than $n$.

(i) Provide the type and implementation of *empty* for the empty binary search tree.

(ii) Provide the type and write a function *lookup* to determine if an element is in a given tree.

(iii) Provide the type and write a function *insert* to insert an element into a given tree.

(iv) Write any functions you may need on binary numbers, making sure they type-check.

(v) Discuss any difficulties or limitations with refinement types you encountered in parts (i)–(iii).

**Exercise 6** Instead of the ordering invariant from Exercise 5 we may wish to verify the balance invariant for the case of AVL trees. Specify a different type of tree with sufficient information to express the balance invariant, write implementations of *lookup* and *insert* (including the necessary rotations to restore the balance invariant) and explore whether the implementation type-checks.

## References

[BC04] Yves Bertot and Pierre Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[Bra13] Edwin Brady. Idris, a general purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CH88] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120, 1988.

[Gir71] Jean-Yves Girard. Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, Amsterdam, 1971.

[LBN17] Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. On subtyping-relation completeness, with an application to iso-recursive types. *ACM Transactions on Programming Languages and Systems*, 39(4):4:1–4:36, March 2017.

[Nor07] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.