ATPLT-2023: Final project report

Implementing Refinement Types

Martin Jensen 201704721
Wolfgang Meier 202202871
Martin Zacho 201507667

December 11, 2023

Abstract

We describe our implementation of type checking for a STLC with refinement types.

Our OCaml implementation follows a tutorial on refinement types [3], which is authored by the main developers of LiquidHaskell [8]. The tutorial starts with a STLC with refinement types and gradually extends it with more interesting features. We extend the STLC with refinement types [3, §3] with inductive data types [3, §7] and recursive functions, which are guaranteed to terminate due to well-founded recursion [3, §4 §9]. We adapt the typing rules from [3, §7] and [3, §9] to adjust for not implementing polymorphism.

The resulting language is expressive enough to encode and machine-check proofs of simple propositions [3, §10].

1 Introduction

Type systems ensure statically the absence of certain classes of errors in a program. They embody the idea that certain operations may only be performed on values of a certain *type*, with the intention that these operations then succeed at runtime. Typically, typing rules are defined on the syntax and restrict the program more than strictly necessar- some programs are rejected that don't lead to failure at runtime. This is unavoidable when capturing non-trivial language properties, as one over-approximates solutions to problems known to be undecidable from Rice's theorem.

Still, many run-time errors aren't caught by type systems of widely used languages, such as that of Java, for instance when indexing arrays out of bounds or dividing by zero. Although many promising results have been achieved in the area of formal methods, proving deep correctness properties of programs (such as in Coq) is a time-consuming and costly affair. Refining types with logical predicates can provide some of the same correctness guarantees for data structures and algorithms, as specifying properties and proving them correct in a proof assistant, but in a way that's more ergonomic and built into the programming language. While the idea goes back at least to the 1970s, it was Freeman and Pfenning who coined the term "refinement types" in [2]. The idea has since been implemented in various languages such as F* [7], Haskell (via LiquidHaskell [8]), Rust (via flux [4]), C (via RefinedC [6]), among many others. This project is based on the topics presented in the refinement types tutorial [3].

2 Syntax and type system

We first describe the syntax and type system of our language on a high level, then we illustrate selected features with examples.

2.1 Syntax

We consider a simply typed lambda calculus with a few extensions, the syntax is given in Figure 1.

STLC The basis of our language is a SLTC in administrative normal form (ANF). In particular, function arguments must be variables, and the let-bound expressions must be simple enough to synthesize a type. Consequently, we use the somewhat verbose notation **let** *one* = 1 **in** *incr one* to adhere to this restriction.

No polymorphism We neither support System F-style type polymorphism nor refinement inference or refinement polymorphism. The tutorial [3] includes these, it could be an interesting extension if given more time.

Inductive data types Our language has support for user defined inductive data types and pattern matching, these are somewhat limited due to the lack of polymorphism. Inductive data types are described in detail in Section 2.3.2.

Well-founded recursion All functions in our language are required to terminate, this is enforced by making use of refinement types. When type checking the body of a recursive function, then, recursive calls are limited to strictly

(Data Constructors)	D ::=	$D1, D2, ...$	
(Type Constructors)	C ::=	$C1, C2, ...$	
(data types)	δ ::=	$\langle C, \overline{D{:}t} \rangle$	
(Terms)	e ::=	c	$c \in \{\text{true}, \text{false}, +, -, 0, 1, -1, \dots\}$
	|	x	
	|	**let** $x = e$ **in** e	
	|	$\lambda x.\ e$	
	|	$e\ x$	
	|	$e{:}t$	
	|	**if** x **then** e **else** e	
	|	**let rec** $x = e{:}t/m$ **in** e	
	|	D	
	|	**switch** $x\ \{\overline{a}\}$	
(Alternatives)	a ::=	$D(\overline{x}) \to e$	
(Termination metrics)	m ::=	p	*decreasing expression*
	|	$p; m$	*lexicographic metric*
(Environments)	Γ ::=	$\emptyset$	
	|	$\Gamma; x{:}t$	
	|	$\Gamma; \delta$	
(Base types)	b ::=	$\mathsf{int} \mid \mathsf{bool}$	
	|	C	
(Kinds)	k ::=	B	*base kind*
	|	$\star$	*star kind*
(Types)	t ::=	$b\{x{:}\ p\}$	*refined basetype*
	|	$x{:}t_1 \to t_2$	*dependent function*
(Predicates)	p ::=	x, y, z	
	|	$\top, \bot$	
	|	$0, -1, 1, ...$	
	|	$p_1 \bowtie p_2$	$\bowtie \in \{+, -, =, \geq, <, ...\}$
	|	$p_1 \wedge p_2$	
	|	$p_1 \vee p_2$	
	|	$\neg p$	
	|	$f(\overline{p})$	*uninterpreted functions*
(Constraints)	c ::=	p	
	|	$c_1 \wedge c_2$	
	|	$\forall x{:}b.\ p \Rightarrow c$	

Figure 1: Syntax

Well-formedness $\boxed{\Gamma \vdash t : k}$

$$\frac{\Gamma; x{:}b \vdash p}{\Gamma \vdash b\{x{:}\ p\} : B}\ (\text{WF-BASE}) \qquad \frac{\Gamma \vdash s : k_s \qquad \Gamma; x{:}s \vdash t : k_t}{\Gamma \vdash x{:}s \to t : \star}\ (\text{WF-FUN})$$

Metric well-formedness $\boxed{\Gamma \vdash m}$

$$\frac{\Gamma \vdash p : int}{\Gamma \vdash p}\ (\text{WFM-BASE}) \qquad \frac{\Gamma \vdash p \qquad \Gamma \vdash m}{\Gamma \vdash p; m}\ (\text{WFM-LEX})$$

Entailment $\boxed{\Gamma \vdash c}$

$$\frac{\mathsf{SmtValid}(c)}{\emptyset \vdash c}\ (\text{ENT-EMP}) \qquad \frac{\Gamma \vdash \forall x{:}b.\ p[x/v] \Rightarrow c}{\Gamma; x{:}b\{v{:}\ p\} \vdash c}\ (\text{ENT-EXT})$$

Subtyping $\boxed{\Gamma \vdash t_1 \prec{:}\ t_2}$

$$\frac{\Gamma \vdash \forall v_1{:}b.\ p_1 \Rightarrow p_2[v_1/v_2]}{\Gamma \vdash b\{v_1{:}\ p_1\} \prec{:} b\{v_2{:}\ p_2\}}\ (\text{SUB-BASE}) \qquad \frac{\Gamma \vdash s_2 \prec{:}\ s_1 \qquad \Gamma; x_2{:}s_2 \vdash t_1[x_2/x_1] \prec{:}\ t_2}{\Gamma \vdash x_1{:}s_1 \to t_1 \prec{:}\ x_2{:}s_2 \to t_2}\ (\text{SUB-FUN})$$

Type synthesis $\boxed{\Gamma \vdash e \Rightarrow t}$

$$\frac{\mathsf{prim}(c) = t}{\Gamma \vdash c \Rightarrow t}\ (\text{SYN-CON}) \qquad \frac{\Gamma \vdash t : k \qquad \Gamma \vdash e \Leftarrow t}{\Gamma \vdash e{:}t \Rightarrow t}\ (\text{SYN-ANN}) \qquad \frac{\Gamma(D) = t}{\Gamma \vdash D \Rightarrow t}\ (\text{SYN-DATA})$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \Rightarrow self(x, t)}\ (\text{SYN-VAR}) \qquad \frac{\Gamma \vdash e \Rightarrow x{:}s \to t \qquad \Gamma \vdash y \Leftarrow s}{\Gamma \vdash e\ y \Rightarrow t[y/x]}\ (\text{SYN-APP})$$

Type checking $\boxed{\Gamma \vdash e \Leftarrow t}$

$$\frac{\Gamma \vdash e \Rightarrow s \qquad \Gamma \vdash s \prec{:}\ t}{\Gamma \vdash e \Leftarrow t}\ (\text{CHK-SYN}) \qquad \frac{\Gamma; x{:}t_1 \vdash e \Leftarrow t_2}{\Gamma \vdash \lambda x.\ e \Leftarrow x{:}t_1 \to t_2}\ (\text{CHK-LAM})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow t_1 \qquad \Gamma; x{:}t_1 \vdash e_2 \Leftarrow t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Leftarrow t_2}\ (\text{CHK-LET})$$

$$\frac{y\ \mathsf{is\ fresh} \qquad \Gamma \vdash x \Leftarrow \mathsf{bool} \qquad \Gamma; y{:}\mathsf{int}\{y{:}\ x\} \vdash e_1 \Leftarrow t \qquad \Gamma; y{:}\mathsf{int}\{y{:}\ \neg x\} \vdash e_2 \Leftarrow t}{\Gamma \vdash \textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Leftarrow t}\ (\text{CHK-IF})$$

$$\frac{t_1 = \overline{y{:}s} \to t_1' \qquad e_1 = \lambda \overline{y}.\ e_1' \qquad \Gamma; \overline{y{:}s}; f{:}lim(\Gamma; \overline{y{:}s}, m, t_1) \vdash e_1' \Leftarrow t_1' \qquad \Gamma; f{:}t_1 \vdash e_2 \Leftarrow t_2}{\Gamma \vdash \textbf{let rec}\ f = e_1{:}t_1/m\ \textbf{in}\ e_2 \Leftarrow t_2}\ (\text{CHK-REC})$$

$$\frac{\Gamma \mid y \vdash a_i \Leftarrow t \quad \mathsf{for\ each\ i}}{\Gamma \vdash \textbf{switch}\ y\ \{\bar{a}\} \Leftarrow t}\ (\text{CHK-SWT})$$

Checking Alternatives $\boxed{\Gamma \mid y \vdash a \Leftarrow t}$

$$\frac{s = \Gamma(y) \qquad \Gamma' = unapply(\Gamma, y, \bar{z}, s) \qquad \Gamma' \vdash e \Leftarrow t}{\Gamma \mid y \vdash D(\bar{z}) \to e \Leftarrow t}\ (\text{CHK-ALT})$$

$$\begin{array}{ll} self(x, b\{v \mid p\}) & = b\{v \mid p \wedge v = x\} \\ self(x, t) & = t \end{array}$$

$$\begin{array}{ll} unapply(\Gamma, y, z; \bar{z}, x{:}s \to t) & = unapply(\Gamma; z{:}s, y, \bar{z}, t[z/x]) \\ unapply(\Gamma, y, \emptyset, t) & = \Gamma; y{:}meet(\Gamma(y), t) \\ \\ meet(b\{v_1{:}p_1\}, b\{v_2{:}p_2\}) & = b\{v_1{:}p_1 \wedge p_2[v_1/v_2]\} \\ meet(x_1{:}s_1 \to t_1, x_2{:}s_2 \to t_2) & = x_1{:}meet(s_1, s_2) \to t' \\ \quad \text{where } t' & = meet(t_1, t_2[x_1/x_2]) \end{array}$$

Figure 2: Typing rules

$$
\begin{array}{ll}
lim(\Gamma, m, t) & = lim'(\Gamma, m, m, t) \\
\\
lim'(\Gamma, m^*, m, x{:}b\{p\} \to t) & \\
\quad \mid \Gamma; x{:}b \vdash m & = x'{:}b\{p'\} \to t[x'/x] \\
\quad \text{where} & \\
\qquad p' & = p[x'/x] \wedge wfr(m^*, m[x'/x]) \\
lim'(\Gamma, m^*, m, x{:}s \to t) & = x'{:}s[x'/x] \to t' \\
\quad \text{where} & \\
\qquad t' & = lim'(\Gamma; x'{:}s, m^*, m[x'/x], t[x'/x])
\end{array}
$$

$$
\begin{array}{ll}
wfr(p, p') & = 0 \leq p' \wedge p' < p \\
wfr(p; m,\ p'; m') & = 0 \leq p' \wedge (p' < p \vee (p' = p \wedge wfr(m, m')))
\end{array}
$$

Figure 3: Typing rules (continued)

smaller arguments w.r.t. a well-founded metric, which also ensures that there is a lower bound. This is described in more detail in Section 2.3.3, for arguments of inductive data types, this allows for specifying e.g. structural recursion.

Predicates Base types can be refined with predicates, which are drawn from the quantifier-free subset of linear arithmetic with uninterpreted functions (called QF-UFLIA in the smtlib standard [1]). The fact, that the logic is decidable, makes type checking for our type system decidable and predictable. If it wasn't decidable, then it may depend on the specific SMT solver (and its heuristics), if a certain term is well-typed, which may not be desirable. The same choice was made for the design of LiquidHaskell [8, §8], type checking in F* however, is undecidable.

2.2 Type system

We implement a bidirectional typing scheme consisting of type checking and type inference ("type synthesis"), the judgements are given in Figure 2.

Type synthesis Simple terms like constants and variables synthesize a type directly, i.e. their type can be inferred from the structure of the term or context Γ. This allows synthesizing precise types, e.g. the refined type $\mathsf{int}\{v\colon v = 42\}$ for a constant 42. When synthesizing a type for a variable, *selfification* [5] is performed on the resulting type of looking up the variable in Γ, i.e. the predicate of the resulting type is extended with an additional equality between the variable and the binder of the predicate. This is required for if statements, where the type of the body depends on the branching condition [3, §4.3.2].

Type checking All terms can be type checked for a provided type, but not all terms synthesize a type. In that case, one can wrap the term with a type annotation. The wrapped term now synthesizes that type, if type checking the original term with the given type succeeds (SYN-ANN). Thus, the notion of bidirectional type checking. Similarly, for let-bindings, the let-bound expression has to synthesize a type, which is assigned to the variable extending Γ before checking the remaining program (SYN-LET).

Type checking a switch term induces one obligation per constructor of the inductive data type that is branched on, an example is described in Section 2.3.2.

Subtyping At some point during the process of type checking, a simple enough subterm will be reached, that a type can be synthesized and shown to be a subtype of the given type (rule CHK-SYN). Subtyping is defined inductively on the type, subtyping for refined base types reduces to showing an implication of the refinement predicates with appropriate substitution. Subtyping on dependent function types is contra-variant in the input types and co-variant in the result type.

Entailment Every subtyping judgement of two base types induces a constraint, which is passed to an SMT solver to ensure it is a tautology. Before that, the formula has to be closed so it does not contain free variables and the refinement predicates of the variables in Γ are included in the formula. The rule ENT-EXT does that, it introduces a $\forall$-quantified implication for every variable from Γ.

2.3 Examples

In this section, we illustrate selected rules of the type system.

To demonstrate how the checking and synthesis judgements go hand in hand, and, what constraints they generate, consider giving the following program the imprecise type $\mathsf{int}\{v\colon v > 10\}$:

$$\textbf{let } x = 12 \textbf{ in } x$$

By the rule CHK-LET it is sufficient to check that

$$\emptyset \vdash 12 \Rightarrow \mathsf{int}\{v\colon v = 12\} \qquad x : \mathsf{int}\{v\colon v = 12\} \vdash x \Leftarrow \mathsf{int}\{v\colon v > 10\}$$

which for the first judgement by SYN-CON means checking that $\mathsf{prim}(12) = \mathsf{int}\{v\colon v = 12\}$. But this is exactly the primitive (and precise) type assigned to the integer literal 12. Now for the second judgement, applying CHK-SYN reduces this to checking that

$$x : \mathsf{int}\{v\colon v = 12\} \vdash x \Rightarrow \mathsf{int}\{v\colon v = 12\} \qquad x : \mathsf{int}\{v\colon v = 12\} \vdash \mathsf{int}\{v\colon v = 12\} \prec\colon \mathsf{int}\{v\colon v > 10\}$$

For synthesizing the type of x, consider for the purpose of this example a simplified version of SYN-VAR, that doesn't perform selfification. This rule then applies, since looking up the type of x in the environment gives exactly $\mathsf{int}\{v\colon v = 12\}$. Now SUB-BASE will reduce the subtyping judgement to checking the entailment

$$x : \mathsf{int}\{v\colon v = 12\} \vdash \forall v : \mathsf{int}.\ v = 12 \Rightarrow v > 10$$

which further reduces by ENT-EXT to

$$\emptyset \vdash \forall x : \mathsf{int}.\ x = 12 \Rightarrow \forall v : \mathsf{int}.\ v = 12 \Rightarrow v > 10$$

which by ENT-EMP reduces to sending the constraint to z3, which will indeed asserted it valid in the theory of arithmetic, since its negation is unsatisfiable (i.e. Z3 is unable to come up with a counter example).

2.3.1 Control flow and path sensitive type checking

To show how our typesystem can give refined types to programs which branches on boolean values, consider the following implementation and application of the function computing the absolute value of an integer:

```
let zero = 0 in
let abs =
  (fn x.
    let b = (lt x) zero in
      if b
      then (sub zero) x
      else x) : x:int → int{v: v ≥ x}
in
let ten = 10 in abs ten
```

We're using the syntax int when the refinement is just $\top$. When binding functions to variables in let expressions, the type of the function must be annotated (using an annotated expression of the form $e : \tau$). Otherwise, synthesizing the let-bound expression will fail. The type we give `abs` ensures that its result will be no smaller than its input. We're using the following specification for `lt`:

$$lt: \quad x{:}\mathsf{int} \to y{:}\mathsf{int} \to \mathsf{bool}\{z\colon (\neg z \vee x < y) \wedge (\neg x < y \vee z\}$$

Therefore, when type checking the `if` expression, the typing context contains the following binding for the value b that is branched on:

$$\Gamma(b) = \mathsf{bool}\{z\colon (\neg z \vee x < zero) \wedge (\neg x < zero \vee z))\}$$

When type checking each branch, we add a binding for a free variable of an arbitrary base type to the typing context, that has been refined to capture the value of b in that branch (by the rule CHK-IF). So we type check $(sub\ zero)\ x$ in the context where $\sigma : \mathsf{int}\{\sigma\colon b\}$ and we type check x in the context where $\sigma : \mathsf{int}\{\sigma\colon \neg b\}$. The variable σ won't ever be used (so its type doesn't matter in CHK-IF), and solely exists so that ENT-EXT will expand it to the SMT constraint $\forall \sigma : \mathsf{int}.\ b \Rightarrow c$, where c is the constraints generated by type checking the first branch (and similarly when $\neg b$ for the second branch). We're giving sub the type $x{:}\mathsf{int} \to y{:}\mathsf{int} \to \mathsf{int}\{v\colon v = x - y\}$, so the first judgement

$$x : \mathsf{int}\{v\colon \top \wedge x = v\}; \sigma : \mathsf{int}\{\sigma\colon b\}; zero : \mathsf{int}\{v\colon v = 0 \wedge zero = v\}; sub : \ldots \vdash (sub\ zero)\ x \Leftarrow \mathsf{int}\{v\colon v \geq x\}$$

in the premises of CHK-IF will produce the constraint

$$\forall \sigma : \mathbb{Z}.\ b \Rightarrow (\forall v : \mathbb{Z}.\ (v = 0 \land zero = v) \Rightarrow \top) \land (\forall v : \mathbb{Z}.\ (\top \land x = v) \Rightarrow \top) \land (\forall z : \mathbb{Z}.\ z = zero - x \Rightarrow z \geq x)$$

saying that the arguments to sub ($zero$ and x) must have subtypes of the types of the formal parameters of sub, and the return type of *sub* must be a subtype of the type we want to give the whole expression.

2.3.2 Inductive data types and pattern matching

We add user-defined inductive data types as a base type of the language. This allows for defining data structures like tuples, lists, trees etc. Due to the lack of polymorphism, the data types are somewhat limited: The programmer must specify the concrete type of the elements of the data type when declaring it, e.g. a list of natural numbers. That is, data types, and types of the language in general, cannot abstract over types, such as in system F_ω. It is nevertheless possible to express some interesting programs. Additionally, the language is extended with a switch expression that allows for destructing a data type value by pattern matching on its constructors.

An inductive data type declaration consists of the name of the data type and a number of data constructors. For example the classic inductive list data type with constructors `Nil` and `Cons`, here limited to elements that are natural numbers:

```
type nlist =
| Nil
| Cons(x:int{0 ≤ x}, xs:nlist)
```

The types of the data constructors are essentially function types, e.g. the `Cons` constructor has the type

$$x{:}\mathsf{int}\{x{:}\ 0 \leq x\} \to xs{:}\mathsf{nlist} \to \mathsf{nlist}$$

This means that constructing a data type reduces to function application (or to a variable for a 0-ary constructors). We also limit the elements of data types to be refined base types, i.e. int, bool or inductive data types.

Before type checking a program, the types of the data constructors of each defined data type are added to the base environment, which means they can be used throughout the entire program. For example, a base environment for a program containing the list data type would be extended in the following way:

$$\Gamma_{\mathsf{base}};\ \mathsf{Nil} : \mathsf{nlist};\ \mathsf{Cons} : x{:}\mathsf{int}\{x{:}\ 0 \leq x\} \to xs{:}\mathsf{nlist} \to \mathsf{nlist}$$

Additionally, it is possible to specify properties of data types using *uninterpreted functions*. For example, we can define the *length* of an nlist with a function $len : \mathsf{nlist} \to \mathbb{N}$, given that we can define it for each constructor. This allows us to reason about the length of lists in the logic, without having to define an actual length function.

type nlist =
| Nil $\to\ \{v : len(v) = 0\}$
| Cons($x : \mathsf{int}\{x{:}\ 0 \leq x\}$, $xs : \mathsf{nlist}$) $\to\ \{v : len(v) = len(xs) + 1\}$

As the dual of data constructor application, switch expressions allow for destructing data type values. Switch expressions take a variable y to be pattern matched, and a list of *alternatives*, of the form $D(\bar{x}) \to e$ where $D(\bar{x})$ is a constructor pattern, and e is the expression to be evaluated if x matches the pattern $D(\bar{x})$.

Checking a switch expression (summarized in the judgement CHK-SWT of 2) against a type t amounts to checking that all the alternatives can be checked against the type t.

To check an alternative $D(x_1, \ldots, x_n) \to e$ against a type t (summarized in the judgment CHK-ALT of 2) is done in two steps

1. The current environment Γ is extended with the pattern binders $x_1, \ldots, x_n$ bound to the types of the parameters of the constructor D. E.g. for the alternative $\mathsf{Cons}(x, xs) \to e$, Γ is extended with $x : \mathsf{int}\{x{:}\ 0 \leq x\};\ xs : \mathsf{nlist}$. Furthermore, the constructor D may refine the constructed value, which means that if y matches the pattern, then it must also have the refinements specified by D, in addition to the refinements of its current type. This is reflected by strengthening the type of y with the additional refinements in the extended environment. E.g. if $\Gamma(y) = \mathsf{nlist}\{y{:}\ p\}$, then in the $\mathsf{Cons}(x, xs)$ branch, y is bound to the type $\mathsf{nlist}\{y{:}\ p \land len(y) = 1 + len(xs)\}$
2. Then, the expression e is checked against t in the extended environment.

Additionally, it is implicitly required that all constructors in the list of alternatives are actually constructors of the type of the variable being pattern matched, and that the list of alternatives is *exhaustive*, as the type system would otherwise be unsound. Due to the lack of polymorphism, subtyping of data types is handled by the subtyping rule for base types.

As an example, consider the program in 4 assuming the definition of `nlist` (2.3.2) which defines a function that safely returns the head of a list by combining pattern matching with data type refinement:

```
let assert = (λb. 0) : b:bool{b: b} → int in
let head =
    (λxs.
        switch xs {
        | Nil              ⇒ assert false
        | Cons(hd, tl)     ⇒ hd
        }) : xs:nlist{xs: 0 < len(xs)} → int
in
...
```

Omitting some details, checking the definition of `head` against its type boils down to checking the validity of the constraint

$$\forall xs : \mathsf{nlist}.\ 0 < len(xs) \wedge 0 = len(ys) \implies \forall b : \mathsf{bool}.\ \neg b \implies b$$

The contradiction $0 < len(xs) \wedge 0 = len(ys)$ in the antecedent is exactly due to the fact that by assumption, the input function input xs must satisfy $0 < len(xs)$, yet while checking the `Nil` branch, xs must also satisfy $0 = len(xs)$, which intuitively means that the branch can never be taken, and thus the type system ensures that `head` can only be applied to a non-empty list. To illustrate this, consider the expression `let xs = Nil in head xs`. Type checking this expression fails, as it yields the constraint

$$\forall \mathsf{Nil} : \mathsf{nlist}.\ len(\mathsf{Nil}) = 0 \implies \forall xs : \mathsf{nlist}.\ xs = \mathsf{Nil} \implies 0 < len(xs)$$

which is not valid due to the equality between xs and Nil (arising from the let-binding) and the congruence property of uninterpreted functions which says that $xs = \mathsf{Nil} \implies len(xs) = len(\mathsf{Nil})$

With the definition of *len*, we can also specify some simple properties of functions on lists, for example that a `map` function returns a list with the same length as the input list, i.e. with the type

$$\mathsf{map} : f{:}(x{:}\mathsf{int} \to \mathsf{int}) \to xs{:}\mathsf{nlist} \to \mathsf{nlist}\{v{:}\ len(v) = len(xs)\}$$

Now, we can verify an implementation of map by checking it against the above type:

```
let rec map =
    (fn f. (fn xs.
        switch xs {
            | Nil              ⇒ Nil
            | Cons(hd, tl) ⇒
                let newhd = f hd in
                let newtl = map f tl in
                Cons newhd newtl
        })
        : f:(x:int → int) → xs:nlist → nlist{v: len(v) = len(xs)} / len(xs)
in
map
```

Listing 1: A definition of a `map` function on lists that preserves length

Introducing a bug in the implementation, for example only returning `newtl`, would result in type checking to fail.

2.3.3 Well-founded recursion: Ensuring termination of recursive functions

Our type system ensures at compile-time that well-typed functions always terminate. The programmer is required to specify a *well-founded* termination *metric* (see fig. 1) for recursive function definitions. A termination metric is either an int-sorted expression in our logic that decreases for each recursive function call and is bounded from below, or a *lexicographic* metric m, n where either m is decreasing, or m is not increasing and n is decreasing. This is reflected in our typing rule CHK-REC, when checking that **let rec** $f = e_1 : x{:}s \to t/m$ **in** e_2 the metric m is then used to *limit* the type $x{:}s \to t$ when type checking the body e_1 of the let-bound function f.

As a simple example, consider the following implementation of the constant zero function for natural numbers:

```
let one = 1 in let rec zero =
    (fn x. let b = (lt x) one in
       if b then 0 else
         let newx = (sub x) one in zero newx)
          : x:int{v: v ≥ 0} → int{v: v = 0} / x
    in let ten = 10 in zero ten"
```

The function is declared to be recursing on strictly smaller values for its argument x, which the type system ensures by giving `zero` the limited type $\sigma{:}\mathsf{int}\{\sigma{:}\ \sigma \geq 0 \wedge \sigma < x\} \to \mathsf{int}\{v{:}\ v = 0\}$, for a fresh variable σ, when type checking the body of the function.

Termination metrics can be arbitrary int-sorted expressions, so we can give a refined type to the structurally recursive append function on nlists, and ensure that it always terminates by specifying the metric $len(xs)$:

```
let rec append =
(fn xs.
  (fn ys.
    switch xs {
    | Nil ⇒ ys
    | Cons(hd, tl) ⇒ let apptl = append tl ys in Cons hd apptl
    }
 : xs:nlist → ys:nlist → nlist{v: len(v) = len(xs) + len(ys)} / len(xs)
in true
```

Since each recursive call is on a sub-structure (the tail) of the input, and values of the data constructors Nil and Cons have been appropriately refined with respect to len, then the body of append can be correctly type checked in the environment where append have been limited to the type

$$\mathsf{append} : \sigma{:}\mathsf{nlist}\{\sigma{:}\ 0 \leq len(\sigma) \wedge len(\sigma) < len(xs)\} \to ys{:}\mathsf{nlist}\{v{:}\ \top\} \to \mathsf{nlist}\{v{:}\ len(v) = len(\sigma) + len(ys)\}$$

The termination metric can also be specified as a *lexicographical* metric, which is a list of smaller metrics. This can be used for termination checking the Ackermann function $\mathsf{ack}(\mathsf{m}, \mathsf{n})$, which does not always recursive on strictly smaller values for any one of its arguments. However, it is terminating since for every recursive call $\mathsf{ack}(\mathsf{m}', \mathsf{n}')$ either $m' < m$ or $m = m' \wedge n' < n$. Its implementation in ANF is

```
let zero = 0 in let one = 1 in
let rec ack = (fn m. (fn n.
  let b = (eq m) zero in
  if b then (add n) one
  else let newm = (sub m) one in
       let b = (eq n) zero in
       if b then (ack newm) one
       else let newn = (sub n) one in
            let ackres = (ack m) newn in
            (ack newm) ackres))
: m:int{v:v≥0} → n:int{v:v≥0} → int{v:v≥0} / m, n in ack
```

and its limited type is

$$\mathsf{ack} : \sigma_m{:}\mathsf{int}\{v{:}\ v \geq 0\} \to \sigma_n{:}\mathsf{int}\{\sigma_n{:}\ 0 \leq \sigma_n \wedge 0 \leq \sigma_m \wedge (\sigma_m < m \vee (m = \sigma_m \wedge 0 \leq \sigma_n \wedge \sigma_n < n))\} \to \mathsf{int}\{v{:}\ v \geq 0\}$$

And our implementation will ensure that it terminates for all $m, n \in \mathbb{N}$.

3 Encoding constraints as SMT queries

We use the official Z3 library for OCaml[1] to interface with Z3, in order to solve constraints generated during type checking and synthesis. In order to check the validity of a closed constraint $c = \forall x.p \Rightarrow q$, we check the satisfiability of the formula $p \wedge \neg q$. If no satisfying assignment exists, i.e. Z3 returns `un-sat`, then c is valid. As the last step of type checking, our implementation transforms all constraints $c_i = \forall x_i.p_i \Rightarrow q_i$ into the formulae $p_i \wedge \neg q_i$, takes the conjunction of all these formulae and asserts that it is closed, then translates it into a Z3.Expr[2] checks whether it is satisfiable, and if so, type checking fails.

The translation into an Z3.Expr is mostly straight forward. Boolean literals, integer numerals, conjunction, disjunction, negation and binary operations of predicates are straight forward to translate. Variables of type `bool` or `int` are similarly translated into variables with their respective Z3 sort, and variables of user defined types are translated into variables of an *uninterpreted* sort, that is initially created for each user defined type using Z3.Sort.mk_uninterpreted[3]. Uninterpreted functions are registered using Z3.FuncDecl.mk_fresh_func_decl[4] and applied using Z3.FuncDecl.apply.

[1] https://z3prover.github.io/api/html/ml/Z3.html
[2] https://z3prover.github.io/api/html/ml/Z3.Expr.html
[3] https://z3prover.github.io/api/html/ml/Z3.Sort.html
[4] https://z3prover.github.io/api/html/ml/Z3.FuncDecl.html

Uninterpreted function inputs/ outputs in Z3 are defined in terms of base sorts, i.e. integers, booleans, or uninterpreted sorts like nlist. Often, we want to express additional restrictions on the co-domain of the functions, e.g. that the co-domain of `len`, the uninterpreted length function on nlists (see listing 2.3.2), is $\mathbb{N}$, in order for it to be used as a termination metric. To enforce these restrictions, the antecedents of all implications in the generated constraints are strengthened, before solving them. For `len`, restricting the co-domain to $\mathbb{N}$ intuitively translates to the universal quantification $\forall l : \mathsf{nlist}.0 \leq len(l)$. To enforce this, all constraints of the form $\forall x : \mathsf{nlist}.\ p \Rightarrow q$ are strengthened to $\forall x : \mathsf{nlist}.\ p \wedge 0 \leq len(x) \Rightarrow q$.

4 Theorem proving

The machinery introduced so far allows proving simple theorems, this works via the Curry-Howard correspondence shown in Table 1. A proposition is encoded as a type, which is then shown to be inhabited by a provided program representing a machine-checkable proof, that the proposition holds.

Proofs	Programs
Propositions	Types
Proofs	Programs
Induction	Recursion
Case distinction	Pattern matching

Table 1: Curry-Howard correspondence

A simple statement one can prove this way is commutativity of addition. This can be encoded as follows.

```
let add_comm =
    (fn x. fn y. 42) : x:int → y:int → int{z: x+y = y+x}
in ...
```

The generated constraint is trivially solved by Z3, as it is a simple fact in linear arithmetic.

$$\forall x, y, z.\ z = 42 \ \Rightarrow\ x + y = y + x$$

We include a demonstration[5] of how one can extend the implementation to support proving more interesting propositions, e.g. the following equality attributed to young Gauss:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

The main ingredient to prove statements like this, is refining the return type of functions with a proposition that reflects the function body, i.e. an equality between the result variable and the embedding of the body into the logic. E.g. constants are directly embedded, and function applications become uninterpreted function calls. This allows doing a 'computation' step of uninterpreted function calls in the logic by rewriting with an equality.

We did not have time to implement and test this embedding, so we specify the embedded type manually and show, how one can use that to type check a simple inductive proof.

Consider the following program that computes the sum of the numbers from 1 to n, the second function represents a proof for the equality.

```
let rec sum =
   (fn n.
      let b = eq n 0 in
      if b
      then 0
      else
        let n1 = sub n 1 in
        let s = sum n1 in
        add s n
   ): n:int{n: n ≥ 0} → int{n: n ≥ 0} / n in

let rec gauss_theorem =
  (fn n.
```

[5] See the branch `proof`, where we extend syntax and logic with multiplication so we can show the correctness of the small Gauss sum.

```
    let b = eq n 0 in

    if b
    then (let t = sum 0 in t)
    else (
      let nm1 = sub n 1 in
      let np1 = add n 1 in

      let t   = gauss_theorem nmo in
      let tt  = sum n in
      let ttt = sum nmo in
      0
    )
) : n:int{n: n ≥ 0} → int{v: 2 * sum(n) = n*(n+1)} / n in

...
```

First, the function `sum` is successfully type checked, then, the result type is refined with the embedding of the body so that the function `gauss_theorem` is type checked under context

$$\Gamma = n{:}\mathsf{int}\{v{:}\ v \geq 0\} \to \mathsf{int}\{v{:}\ v \geq 0 \land v = sum(n) \land (n = 0 \Rightarrow v = 0\ \land\ n \neq 0 \Rightarrow v = n + sum(n-1))\}$$

where the second conjunct is equivalent to the embedding of sum's function body, that was simplified for readability.

Our implementation type checks the function `gauss_theorem` successfully in isolation when provided the context Γ. The structure of the proof is determined by the `sum` program, the if-statement does a case distinction on n, if it is equal to 0, the first implication (in the result type of `sum`) solves the goal. For the case where $n > 0$, it suffices to call *sum n* and *sum (n-1)*, and, mainly, the function `gauss_theorem` recursively, which corresponds to applying the induction hypothesis.

Here, it is crucial, that recursive calls are limited by the well-founded metric. Otherwise, one could trivially proof `False` by a recursive call with the same argument.

5 Evaluation and Discussion

In this section, we evaluate our implementation and discuss ways to improve it.

Unit testing We check, that our implementation is sensible using a test-suite consisting of programs of our own as well as examples from [3]. Some of the examples are: sum of nats, a tail-recursive sum of nats, length of lists, append of lists, foldl specialized to nlists to compute the sum of nats, range of nats to nlist, Ackermann and braid of lists. This does not guarantee correctness, of course, as testing cannot show the absence of bugs.

Lack of polymorphism We're able to express a wide range of pure data structures and programs, the lack of type polymorphism is definitely a drawback. However, it was a deliberate choice not to add that to the language (yet), since it complicates the constraints and in particular the debugging process. With more time, this could have been interesting to add.

Encode disjointness of data constructors in logic In our implementation of inductive data types, constructors are not disjoint by default, as we don't encode it in the type. Specifying distinguishing properties like *len* on nlist effectively enforce the disjointedness of Nil and Cons, and it can also be done manually, like in the RGB (Red, Green, Blue) tests in our test suite where we add a predicate e.g. *isRed* as an uninterpreted function, however it could be built into the language to make it more ergonomic. The disjointedness is important for path-sensitive reasoning in the branches of `switch` expressions.

To support automatically discriminating between constructors of a data type, for each user defined type T with data constructors $\mathsf{D}_1, ..., \mathsf{D}_n$, one could automatically add uninterpreted functions $\mathsf{isD_i} :\ \mathsf{T} \to \{\top, \bot\}$, i.e. predicates for each data constructor $\mathsf{D_i}$, that will then be used to refine values v of type T so $\mathsf{isD_i}(v)$ if v has been constructed by $\mathsf{D_i}$ and $\neg\mathsf{isD_i}$ otherwise.

Extending proving capabilities With more time, it would be interesting to turn our system into a proper proof assistant. For that, the main thing missing is the embedding of the function body into the result type as described in Section 4. To make writing proofs easier, better error messages are also required, in particular, splitting up the constraint in multiple smaller ones. E.g. for a proof by case distinction, it would be of great use to report to the user, which case already holds and which one is left to prove. This can be achieved by checking the `then` and `else` branches of if-statements separately.

Annotation overhead Another possible extension, given more time, would be to implement inference of refinements. This would help cut down the amount of refinement annotations needed on functions, and make them more re-usable. As an example, consider the program in listing 2.

```
let inc = (λx. add x 1) : x:int → int{v: v = x + 1} in
let apply = (λf. (λx. f x)) : f:(x:int → int) → x:int → int in
let x = 41 in
apply inc x
```

Listing 2: A general definition of an `apply` function and example usage

Intuitively, the program should type check with the type $\mathsf{int}\{v\colon v = 42\}$. However, type checking fails, because at the function application `apply inc x`, the only thing that is known about `apply` is that it takes a function $f : x{:}\mathsf{int} \to \mathsf{int}$, and an input $x : \mathsf{int}$ and returns an int, and so the refinements on the output of the concrete input `inc` are "lost", in a sense. We can remedy this by changing the type of `apply` to restrict the input function `f`, and a refinement of the output, as shown in listing 3

```
let inc = (λx. add x 1) : x:int → int{v: v = x + 1} in
let apply = (λf. (λx. f x)) : f:x:int → int{v: v = x + 1} → x:int → int{v: v = x + 1} in
let x = 41 in
apply inc x
```

Listing 3: A definition of `apply` with additional refinements

Unfortunately, this means that we now cannot use a decrement function $\mathsf{dec} : x{:}\mathsf{int} \to \mathsf{int}\{v\colon v = x - 1\}$ with `apply` to type check `apply dec 42` against the type $\mathsf{int}\{v\colon v = 41\}$. While this is a somewhat contrived example, the point is that there is a trade-off between how 'general' functions can be, and how much can be verified at specific function application sites. This is more of a practical consideration, since it is always possible to just duplicate functions to support different usages.

Unhelpful error messages Since one big constraint is generated for the entire program, it can be difficult to determine the issue, if type checking fails. Inspecting the counter examples produced by Z3 manually is quite cumbersome.

It is possible to provide more precise error messages by solving more smaller constraints, e.g. separate constraints for the `then` and `else` branches of an if-statement. This would help pinpoint the location and allow showing the user better error messages. The decreased performance would not be an issue for the size of programs we're considering.

Note, that showing useful error messages is a challenge in general for systems where constraints are collected and solved. We can, however, do much better than is currently implemented.

6 Conclusion

We presented our OCaml implementation of a refinement type checker for a SLTC with if-statements, switch expressions, recursive functions and user defined algebraic data types (albeit without type polymorphism), termination is ensured by well-founded termination metrics. Refinements are predicates from a quantifier free, decidable logic, the generated constraints are solved by Z3.

Our approach follows [3], we just adapted the rules in the later chapters to account for the lack of type polymorphism. We tested our implementation using a test suite which includes many of the examples from [3] including termination checking the Ackermann function.

The introduced machinery allows machine-checking the proofs of simple propositions via the Curry-Howard correspondence, we described a basic prototype implementation for that.

A Practical guide

Type checking is implemented in OCaml, generated constraints are passed to Z3. The accompanied code comes with a nix-based setup using the package versions of NixOS 23.05 (May 2023).

We provide a test suite of around 130 tests for our implementation [6], which can be run by typing `make test`.

For demonstration purposes, we also provide a few selected example programs[7], which can be run with e.g. `make demo-42`, as shown in Figure 4.

```
[nix-shell:~/]$ make demo-42

dune exec _build/default/bin/main.exe -- -example 42 -debug
Done: 23% (29/125, 96 left) (jobs: 0)
Using test program: 42
Generated constraint:

(forall v:Z. v=42 -> v=42)

Z3: success! Tautology.
```

Figure 4: Type checking the constant 42

References

[1] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The smt-lib standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[2] Tim Freeman and Frank Pfenning. "Refinement types for ML". In: *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 1991, pp. 268–277.

[3] Ranjit Jhala, Niki Vazou, et al. "Refinement types: A tutorial". In: *Foundations and Trends® in Programming Languages* 6.3–4 (2021), pp. 159–317.

[4] Nico Lehmann et al. *Flux: Liquid Types for Rust*. 2022. arXiv: 2207.04034 [cs.PL].

[5] Xinming Ou et al. "Dynamic typing with dependent types". In: *Exploring New Frontiers of Theoretical Informatics: IFIP 18th World Computer Congress TC1 3rd International Conference on Theoretical Computer Science (TCS2004) 22–27 August 2004 Toulouse, France*. Springer. 2004, pp. 437–450.

[6] Michael Sammler et al. "RefinedC: Automating the foundational verification of C code with refined ownership types". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 158–174.

[7] Nikhil Swamy et al. "Secure distributed programming with value-dependent types". In: *ACM SIGPLAN Notices* 46.9 (2011), pp. 266–278.

[8] Niki Vazou. *Liquid Haskell: Haskell as a theorem prover*. University of California, San Diego, 2016.

[6] See in particular the file *lib/typecheck_test.ml*

[7] See the Makefile