Rich types, Tractable typing – Functional Correctness –

Yann Régis-Gianas
yrg@irif.fr

2019-01-11

Plan

From λLF to a dependently typed programming language

Cayenne

Dependent ML

Refinement types

Dependent Haskell

Higher-order Hoare Logic

Towards a dependently-typed programming language

There are a lot of active research on the metatheory of Pure Type Systems [1] because, despite of their apparent simplicity, a lot of technical details are very hard to figure out. As a consequence, extending them to handle **stateful computation or non termination**, is a challenge.

Currently, in the programming language community, this extension is considered from two different points of view:

- Curry-Howard centric: [**ynot08**, 2, 3]
  Extend the logic to cope with side-effects.
  What is a program with side-effects proving?
- Programming centric [4–6]:
  Assume a programming language with side-effets. How to prevent the impure terms to pollute the (logical) types?
  (This is required for the decidability of type-checking.)

Yann Régis-Gianas
yrg@irif.fr

Plan

From λLF to a dependently typed programming language

Cayenne

Dependent ML

Refinement types

Dependent Haskell

Higher-order Hoare Logic

Cayenne[4]

Cayenne is a dependently-typed programming language introduced in 1998 by Augustsson. The only side-effect of Cayenne is non termination, but it is still problematic since the type-checker can enter an infinite loop. This design choice has several consequences:

- The type system is an **inconsistent logic**. This does not mean that the language is unsafe[7]. Hence, this design choice restricts the ambition of the programming language: it is not to be used as a proof assistant.
- The typechecker is **more difficult to use** since the programmer has to figure out by himself if type-level terms are diverging or not.
- Even though types can be easily erased, **erasure of purely static terms is a challenging problem**. Indeed, determining if a term only contributes to type-level computation is undecidable in general.

Three type-centric approaches, and another.

Several proposal have been made to circumvent these problems. We will have a (quick) look at the Dependent ML [8] approach, at the Refinement Types [9] and at the ongoing work about Dependent Haskell [6]. Finally, we will present an Higher-Order Hoare Logic which take some distance from the type-centric approaches.

Yann Régis-Gianas
yrg@irif.fr

Plan

Yann Régis-Gianas
yrg@irif.fr

Restricting type families

A direct way of preventing impure terms to flow into types is **to restrict the syntax** of type family application to have indexes only range in a set of pure terms.

What are these pure terms exactly? These are terms that are harmless to the decidability of type-checking. Non terminating terms are clearly impure. How to forbid them syntactically in types while preserving the substitutivity of typing?

From that perspective, the application rule is the most problematic in λLF:

$$\frac{\Gamma \vdash t_1 : \forall(x : \tau_1).\tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : [x \mapsto t_2]\tau_2}$$

How to restrict the allowed t_2 here?

Restricting type families

A direct way of preventing impure terms to flow into types is **to restrict the syntax** of type family application to have indexes only range in a set of pure terms.

What are these pure terms exactly? These are terms that are harmless to the decidability of type-checking. Non terminating terms are clearly impure. How to forbid them syntactically in types while preserving the substitutivity of typing?

From that perspective, the application rule is the most problematic in λLF:

$$\frac{\Gamma \vdash t_1 : \forall (x : \tau_1).\tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : [x \mapsto t_2]\tau_2}$$

How to restrict the allowed t_2 here? Forbid functions! They may hide non terminating terms!

Yann Régis-Gianas
yrg@irif.fr

A syntactic class for type family index

Dependent ML [8] introduces two distinct applications and abstractions.

The first, dependent, application forces the argument to have an **index sort** I:

$$\begin{array}{rcl} I & ::= & \mathsf{int} \mid \{x : \mathsf{int} \mid P\} \\ P & ::= & P \wedge P \mid i \leq i \\ i & ::= & x \mid q \mid q.i \mid i + i \\ & & \text{where } q \in \mathbb{Z} \end{array}$$

The syntax for terms is:

$$t ::= x \mid \lambda(x : I).t \mid \lambda(x : \tau).t \mid t\ [i] \mid t\ t$$

Any impure constant can be added at the level of terms (fixpoint combinator, references, ...) as long as it is not present in types.

Type-checking in DML

The dependent products are restricted to quantification over index sorts:

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \forall(x : I).\, \tau \mid \tau[i] \mid \tau \to \tau \\ K & ::= & \star \mid \forall(x : I).\, K \end{array}$$

Most of the rules can be imported from simply typed λ-calculus. An interesting one deals with application of an index to a term:

$$\frac{\Gamma \vdash t : \forall(x : I).\tau \qquad \Gamma \models i : I}{\Gamma \vdash t[i] : [x \mapsto i]\tau}$$

In order to validate $\Gamma \models i : I$, a solver for linear arithmetic is necessary.

How to relate dynamic terms to static values?

In this setting, integers are only present at the level of index sorts. In order to handle runtime integers, another syntactic class of terms must be introduced.

$$t \quad ::= \quad \ldots \mid k \mid \ldots$$
$$\text{with } k \in \mathbb{Z}$$

As with GADTs, we relate these runtime integers with static integers using **singleton types** of the form "$Int(i)$", representing for each static integer i, the unique dynamic representation of i.

How to populate typing environments with assumptions about these integers?

How to relate dynamic tests to static assumptions?

When we are dynamically comparing an integer with another one, it would be nice to have the type system aware of the conclusion of this test.

It is possible to define special booleans tailored to that purpose:

$$\begin{aligned}
\mathsf{true} &: \forall(x : \{x : \mathsf{int} \mid 1 \leq x\}).\ \mathsf{Bool}\ (x) \\
\mathsf{false} &: \forall(x : \{x : \mathsf{int} \mid x \leq 0\}).\ \mathsf{Bool}\ (x) \\
\mathsf{leq} &: \forall(x, y : \mathsf{int}).\ \mathsf{Int}\ (x) \to \mathsf{Int}\ (y) \to \mathsf{Bool}\ (1 + y - x)
\end{aligned}$$

Then, it is straighforward to devise a special conditional expression to lift dynamic information to the static world:

$$\frac{\Gamma \vdash t_1 : \mathsf{Bool}\ (i) \qquad \Gamma, 1 \leq i \vdash t_2 : \tau \qquad \Gamma, i \leq 0 \vdash t_3 : \tau}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : \tau}$$

Benefits of DML

Even if DML only focuses on a very restricted set of pure terms, a lot of useful properties can be internalized inside the equivalence relation that are hard to handle with a definitional equality.

For instance, the commutativity of the addition or the neutrality of 0 with respect to addition are automatically taken into account by the type-checker.

Example (in an imaginary O'Caml with dependent types)

If we come back to the example of concatenation between two vectors indexed by their lengths. We have seen that a dependently typed programming language can type-check the following program:

```
1  let rec concat
2  : forall 'a. forall (n m : nat).
3    vector n 'a -> vector m 'a -> vector (n + m) 'a
4  = fun n m v1 v2 ->
5    match v1 with
6    | Nil -> v2
7    | Cons [h] (x, xs) -> Cons [h + m] (x, concat h xs v2)
```

because definitional equality is able to show that $0 + n$ is convertible to n and $succ\ (h + m)$ is convertible to $(succ\ h) + m$. Yet, does it still work if we had done the recursion over `v2` instead of `v1`?

Yann Régis-Gianas
yrg@irif.fr

Example (in an imaginary O'Caml with dependent types)

No!

```
1  let rec concat
2  : forall 'a. forall (n m : nat).
3    vector n 'a -> vector m 'a -> vector (n + m) 'a
4  = fun n m v1 v2 ->
5    match v2 with
6    | Nil -> v2
7    | Cons [h] (x, xs) -> Cons [h + m] (x, concat h xs v2)
```

because this time, $n + 0$ is not convertible n neither $succ\ (n + h)$ is convertible to $n + (succ\ h)$.

In DML, there is no distinction between the provability of $n + 0 = n$ and $0 + n = n$. Thus, the two versions of this program would be well-typed.

Yann Régis-Gianas
yrg@irif.fr

Extending the DML approach

The fundamental idea behind Dependent ML is the respect of the **phase distinction**, a clear separation between a compile-time and a run-time. The (so-called) dynamic terms of the programming language and the static terms of the language of types are syntactically disjoint. Singleton types are the only way to relate a static terms to a dynamic terms.

The ATS [5] programming language generalizes Dependent ML with GADTs, which makes possible the **dynamic representation of proof evidence** that some property over static values is satisfied. Eventually, ATS ends up to be both a programming language with effectful operations and a theorem proving language manipulating proof terms. This idea is also used in the Concoqtion language, a mix of O'Caml and Coq[10].

CoqMT [11] is an implementation of Coq that internalizes decision procedures for first order theories in the conversion rule, which makes more dependently-typed programs typable.

Weakness of ATS

The idea of preserving phase distinction thanks to a syntactic separation between static and dynamic terms allows effectful computations in the programming language.

However, this feature implies a syntactic heaviness because, for each runtime formal argument, two abstractions are needed to introduce both the runtime value and its static counterpart:

```
1  (* In ATS, ``int'' is both a sort and a type constructor *)
2  (* of type ``int -> type''. *)
3  typedef Nat = [n:int | n >= 0] int n
4  (* [n:int | n >= 0] is an existential quantification. *)
5
6  (* ``nat'' is the sort [n:int | n >= 0]. *)
7  fun fact2 {n:nat} (x: int n): Nat =
8    if x > 0 then x nmul fact2 (x-1) else 1
9  (* assuming nmul has type ``(Nat, Nat) -> Nat''. *)
```


Yann Régis-Gianas
yrg@irif.fr

Plan

Refinement types

Let us have the look at the exam of 2014!

In a nutshell

The main idea of refinement types based system is to use a rule of the form:

$$\frac{\Gamma \vdash t \Rightarrow t'}{\Gamma \vdash \{x : \tau \mid t\} <: \{x : \tau \mid t'\}}$$

where t and t' are two programs that denote properties and the judgment in hypothesis is referring to the semantics of t and t', namely that in every context where t evaluates to **true**, so does t'. This proof can be delegated to some program logic.

In practice though, these systems are usually equipped with a type-and-effect inference system to check that t and t' are total (and pure) terms.

In the last lecture, we will see that F* is generalizing this idea to reason about effectful programs using so-called Dijsktra monads.

Plan

Yann Régis-Gianas
yrg@irif.fr

Dependent Haskell

A recent line of work[6] is trying to conservatively extend Haskell with dependent types. More precisely, the first step that has been achieved is a replacement of F_C, the core language of GHC with DC an extension of F_C with dependent types.

The system and its metatheory are too technical to fit in this slide, but let us note these three ideas:

- The system is inspired by the **Calculus of Implicit Construction**[**barras**, 2]: annotations for implicit terms that must be erasable.
- Equality between types are explicitly justified by coercion proof terms which must be provided by programmers. In other words, (some part of) convertibility is delegated to the programmers.
- Equality between types are not types.

About type-centric approaches

DML, F* and DC are **complex** type systems, made of hundred of inference rules, which interactions are subtle. In that situation, mechanized proofs of soundness seem mandatory!

But, remember that the existence of a mechanized proof is not the final argument to trust a formal development: the formal definitions, even mechanized, are in the trusted base. Hence, **simplicity** of these definitions is an important criteria to consider too.

Plan

Yann Régis-Gianas
yrg@irif.fr

Hoare logic based proof systems

The seminal work of Floyd, Hoare and Dijsktra [12] in the sixties was focused on proving the correctness of programs with respect to their specifications. To that purpose, Hoare triple were introduced as a judgment describing how a command C transforms the machine state.

$$\{P\}\ C\ \{Q\}$$

is read as a partial correctness statement:

> "For all state such that the precondition P holds before the execution of the command C, the postcondition Q will hold after the execution of the command C if C terminates."

During forty years, the initial proof system of Hoare has been extended towards several directions: total correctness, concurrency, object-orientation...

Verification-condition generation

Hoare's proof system can be used to reduce the correctness of a program with respect to a specification to the validity of a set of formulæ. There are many ways of generating this set of proof obligations. Maybe the most common is the weakest precondition generation which computes the weakest condition such that a program p satisfies a postcondition Q.

Given the program "$x := 0; x := !x + 1$" and the postcondition $x > 0$, we get:

$$\{?\}\ x := 0\ \{?\}\ x := !x + 1\ \{x > 0\}$$

Verification-condition generation

Hoare's proof system can be used to reduce the correctness of a program with respect to a specification to the validity of a set of formulæ. There are many ways of generating this set of proof obligations. Maybe the most common is the weakest precondition generation which computes the weakest condition such that a program p satisfies a postcondition Q.

Given the program "$x := 0; x := !x + 1$" and the postcondition $x > 0$, we get:

$$\{?\}\ x := 0\ \{x + 1 > 0\}\ x := !x + 1\ \{x > 0\}$$

Verification-condition generation

Hoare's proof system can be used to reduce the correctness of a program with respect to a specification to the validity of a set of formulæ. There are many ways of generating this set of proof obligations. Maybe the most common is the weakest precondition generation which computes the weakest condition such that a program p satisfies a postcondition Q.

Given the program "$x := 0; x := !x + 1$" and the postcondition $x > 0$, we get:

$$\{0 + 1 > 0\}\ x := 0\ \{x + 1 > 0\}\ x := !x + 1\ \{x > 0\}$$

...which means that the validity of the formula "$0 + 1 > 0$" implies the postcondition of this program. Hopefully, this formula can be automatically discharged by any basic decision procedures aware of basic arithmetic.

Hoare logic and programming languages

Many Hoare-logic based systems have been developed over the years [13, 14]. Thanks to the huge advances in automatic proving, they have shown promising results to scale up program verification.

Yet, these systems primarily focused on imperative programming languages. Thus, a lot of effort has been made to deal with programs modifying heap-based data structures. For example, separation logic has been devised to easily maintain non aliasing invariants. Indeed, the proof of a simple program like the reversal of a linked list is not immediate.

The reversal of a linked list in Caduceus

```
/*@ requires is\_list(p0)]
  @ ensures \forall plist l0; \old(llist(p0, l0))
            => llist(\result, rev(l0)) @*/
list rev(list p0) {
  list r = p0; list p = NULL;
  /*@ invariant
        \exists plist lp; \exists plist lr;
          llist(p, lp) && llist(r, lr) && disjoint(lp, lr) &&
          \forall plist l;
            \old(llist(p0, l)) => app(rev(lr), lp) == rev(l)
    @ variant length(r) for length_order */
  while (r != NULL) {
    list q = r;
    r = r->tl;
    q->tl = p;
    p = q;
  }
  return p;
}
```

`llist` is a binary predicate that relates a pointer to a logic list: there is a path in the heap from this pointer that is a low-level representation for this list.

A Hoare-logic for functional programs[15]

Functional programs are easier to prove than imperative ones because they are working on high-level data structures defined by inductive types instead of an heap-allocated global graph. Indeed, inductive types provide a nice level of abstraction because of their induction scheme that can be applied to prove a program that manipulates them.

Yann Régis-Gianas
yrg@irif.fr

A Hoare-logic for functional programs[15]

```
1  let rec tr_rev accu l =
2  match l with
3  | [] -> accu
4  | x :: xs -> tr_rev (x :: accu) xs
```

What would be the logic assertions for this program?

A Hoare-logic for functional programs[15]

```
1  let rec tr_rev accu l
2  returns l'
3  ensures (l' = rev l @ accu)
4  = match l with
5  | [] -> accu
6  | x :: xs -> tr_rev (x :: accu) xs
```

What remains to be proved?

Yann Régis-Gianas
yrg@irif.fr

A Hoare-logic for functional programs[15]

```
1  let rec tr_rev accu l
2  returns l'
3  ensures (l' = rev l @ accu)
4  = match l with
5  | [] -> accu
6  | x :: xs -> tr_rev (x :: accu) xs
```

What remains to be proved?

(1) `l = [] ⊨ accu = rev l @ accu`

(2) `l = x :: xs, l'' = rev xs @ (x :: accu)`
`⊨ l'' = rev (x :: xs) @ accu`

How do specifications for functional programs look like?

As we have explained, an Hoare logic for functional programs is not concerned with state related issues like “How to specify and prove aliasing properties?” or “How to reason about the internal state of a module or an object?”. These are orthogonal (yet interesting) issues.

Functional programming is all about polymorphism, type abstraction and modularity. In particular, modularity is achieved thanks to reusable higher-order functions. How should we specify such a general functions?

An higher-order function

Here is a function that performs a linear search through a list to determine if there exists an element of this list that satisfies a predicate implemented by a boolean function:

```
let rec list_exists (f, l) where (∀ x. x ∈ l ⇒ pre (f) (x))
returns b
where (b = true ⇒ ∃ x. x ∈ l and post (f) (x) (true))
   and (b = false ⇒ ∀ x. x ∈ l ⇒ post (f) (x) (false))
= match x with
  | [] → false
  | x :: xs → f (x) ∨ list_exists (f, xs)
  end
```

The specification of `list_exists` is referring to the unknown specification of `f` using the operators `pre` and `post`.

Let's design a Hoare logic for Higher-Order programs

To design an Hoare logic based system, one must answer the following questions:

- Which logic should be used to write the assertions?
- How to connect the program source code and these assertions? In particular, how will the logic refer to the computation results?
- Does it work in practice? Are the common proof obligations easily managable using an automatic prover?

The logic is Higher-Order

A **higher-order** logic seems to be the most natural choice. Indeed, since higher-order programs are parameterized by programs, higher-order specifications should be parameterized by specifications.

```
let rec list_exists (f, l) where (∀ x. x ∈ l ⇒ pre (f) (x))
returns b
where (b = true ⇒ ∃ x. x ∈ l and post (f) (x) (true))
   and (b = false ⇒ ∀ x. x ∈ l ⇒ post (f) (x) (false))
= match x with
   | [] → false
   | x :: xs → f (x) ∨ list_exists (f, xs)
   end
```

The logic is polymorphically typed

To specify programs written in a polymorphic programming language, the logic must be polymorphic too. As a consequence, we assume a quantification over types in our logic.

- A family of properties indexed by a type variable can be defined :

$$\forall\alpha.\forall(x,y:\alpha).\mathsf{r}\,\alpha\;x\,y \Leftrightarrow \mathsf{r}\,\alpha\;y\,x$$

- ...and instantiated to ground types :

$$\forall(x,y:int).\mathsf{r}\,int\,x\,y \Leftrightarrow \mathsf{r}\,int\,y\,x$$

(In the following, an explicitly typed language is used to ease the presentation but Hindley-Milner type inference is implemented in practice.)

Functions are first class values

What are the value of a purely functional programming language?

$$
\begin{array}{rcll}
v & ::= & x\ \overline{\tau} & \text{Variable} \\
& | & K\ \overline{\tau}(v, \ldots, v) & \text{Data} \\
& | & \textsf{fun}\ f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = t & \text{Recursive function}
\end{array}
$$

The body t of a recursive function allows **unbounded recursion**. These kind of values cannot be directly reflected as a term of the higher-order logic because all these logical terms are strongly normalizing.

How can a proof system handle such a hazardous object?

(i) By using a logic that deals with non-termination.
(ii) By restricting the programming language to be strongly normalizing.

$\Rightarrow$ A third way is possible ...

Getting rid of the function's code, keeping its specification

It is possible to design a reflection of values at the logical level that annihilates exactly the hazardous part of the values while still allowing the characterization of computational results in specifications.

The logic type $\lceil \tau \rceil$ reflects the computational type τ in the logic.

$$\begin{array}{rcl} \lceil \alpha \rceil & = & \alpha \\ \lceil \varepsilon\ \overline{\tau} \rceil & = & \varepsilon\ \lceil \overline{\tau} \rceil \\ \lceil \tau_1 \dot{\to} \tau_2 \rceil & = & (\lceil \tau_1 \rceil \to \mathsf{prop}) \times (\lceil \tau_1 \rceil \to \lceil \tau_2 \rceil \to \mathsf{prop}) \end{array}$$

The logic term $\lceil v \rceil$ reflects the computational value v in the logic. A function is reflected by a **pair formed by its precondition and its postcondition**.

$$\begin{array}{rcl} \lceil x\ \overline{\tau} \rceil & = & x\ \lceil \overline{\tau} \rceil \\ \lceil D\ \overline{\tau}\ v_1, \ldots, v_n \rceil & = & D\ \lceil \overline{\tau} \rceil\ \lceil v_1 \rceil, \ldots, \lceil v_n \rceil \\ \lceil \mathsf{fun}\ f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = t \rceil & = & (\lambda(x_1 : \lceil \tau_1 \rceil).F_1,\ \lambda(x_1 : \lceil \tau_1 \rceil).\lambda(x_2 : \lceil \tau_2 \rceil).F_2) \end{array}$$

Thus, the precondition operator **pre** is π_1, the postcondition operator **post** is π_2.

Yann Régis-Gianas
yrg@irif.fr

Syntax of the programming language

Expressions

$$
\begin{array}{rcll}
t & ::= & v & \text{Value} \\
 & | & v\ v & \text{Application} \\
 & | & \mathsf{let}\ (x\,\overline{\alpha} : \tau / F) = t\ \mathsf{in}\ t & \text{Local binding} \\
 & | & \mathsf{case}\ v\ \mathsf{of}\ c & \text{Pattern matching}
\end{array}
$$

Cases

$$
\begin{array}{rcll}
c & ::= & \emptyset & \text{Nil} \\
 & | & p \Rightarrow t \mid c & \text{Cons}
\end{array}
$$

Patterns

$$
\begin{array}{rcll}
p & ::= & x\,\overline{\tau} & \text{Variable} \\
 & | & D\,\overline{\tau}\ p, \ldots, p & \text{Data}
\end{array}
$$

The terms of this functional language are in A-normal: every intermediate value that is met during evaluation is given a name by construction. It is easy to translate a term in the standard syntax into this form. Notice that every `let` is annotated by a logic assertion.

How to generate proof obligations?

The logic assertions are shown to be correct thanks to a condition-verification generation algorithm. It works by propagating postcondition top-down, that is inside the body of functions and on the left-hand side of `let` while populating the assumptions with the precondition and the statically known consequences of the operational semantics.

For instance, in the following term, "sorted l" is propagated into the two branches of the conditional and the fact that "c = true" is assumed in the first branch whereas "c = false" is assumed in the second one:

$$\textbf{let}\ \mathsf{l}\ \textbf{where}\ \mathsf{sorted}\ \mathsf{l} = \\ \textbf{if}\ \mathsf{c}\ \textbf{then}\ \mathsf{e}_1\ \textbf{else}\ \mathsf{e}_2$$

Only 5 syntax directed rules define this propagation for the entire language!

Judgments

- Well-typed values:

$$\Gamma \vdash v : \tau$$

- Valid formulae:

$$\Gamma \models F$$

  (produces a proof obligation)

- Well-typed terms:

$$\Gamma \vdash t : \tau \, \{P\}$$

  (P always has the type $\lceil \tau \rceil \to \mathsf{prop}$)

The condition-verification generation is defined by the third judgment: the leaves of the proof trees for this judgment will consist in judgments of the second form, that correspond to proof obligations. Well-typedness is a side condition that is necessary to define logical reflection and to ensure safety.

Typing rule of values

As values can be reflected on the logical side, a postcondition can be proved by directly generating a proof obligation:

$$\text{Value}\ \frac{\Gamma \vdash v : \tau \qquad \Gamma \models P\ \lceil v \rceil}{\Gamma \vdash v : \tau\ \{P\}}$$

Examples:

$$\frac{\Gamma \vdash 2 : \mathsf{int} \qquad \Gamma \models \mathsf{even}\ 2}{\Gamma \vdash 2 : \mathsf{int}\ \{\mathsf{even}\}}$$

Typing rule for application

To check that an application is well-formed, one must check that the precondition of the function computed by the left hand side is valid. Then, the postcondition of this function can be used to prove the validity of the expected postcondition.

App

$$\frac{\Gamma \vdash f : \tau_1 \dot{\to} \tau_2 \qquad \Gamma \vdash arg : \tau_1 \qquad \Gamma \models \mathsf{pre}(\lceil f \rceil)\ \lceil arg \rceil \qquad \Gamma \models \mathsf{post}(\lceil f \rceil)\ \lceil arg \rceil \Rightarrow P}{\Gamma \vdash f\ arg : \tau_2\ \{P\}}$$

($P_1 \Rightarrow P_2$ is a shortcut for $\forall (res : \lceil \tau_2 \rceil).P_1\ res \Rightarrow P_2\ res$.)

Example:

$$\frac{\begin{array}{c}\Gamma \vdash \text{list\_exists} : (\mathbf{int} \to \mathbf{bool}) \to \text{list}\ \mathbf{int} \to \mathbf{bool} \\ \Gamma \vdash [1;2] : \text{list}\ \mathbf{int} \qquad \Gamma \vdash \text{is\_even} : \mathbf{int} \to \mathbf{bool} \\ \Gamma \models \mathbf{pre}(\text{list\_exists})\ \lceil \text{is\_even} \rceil, [1;2] \\ \Gamma \models \forall b.\mathbf{post}(\text{list\_exists})\ \lceil \text{is\_even} \rceil, [1;2]\ b \Rightarrow b = \mathbf{true}\end{array}}{\Gamma \vdash \text{list\_exists is\_even}, [1;2] : \text{bool}\ \{\lambda b.b = \mathbf{true}\}}$$

Typing rule for local binding

The `let` construct acts as a cut: it introduces an intermediate step in the proof by asserting a property on a value, proving it and then assuming it in the sequel.

Let

$$\frac{\Gamma, \overline{\alpha} \vdash t_1 : \tau_1 \ \{\lambda(x : \lceil \tau_1 \rceil).F\} \qquad \Gamma, (x : \forall \overline{\alpha}.\, \tau_1), \forall \overline{\alpha}.[x \mapsto x\ \overline{\alpha}]F \vdash t_2 : \tau_2 \ \{P\}}{\Gamma \vdash \mathsf{let}\ (x\ \overline{\alpha} : \tau_1 / F) = t_1\ \mathsf{in}\ t_2 : \tau_2 \ \{P\}}$$

Example:

$$\frac{\Gamma \vdash 2 : \mathsf{int}\ \{\lambda(x : \mathsf{int}).\mathsf{even}(x)\} \qquad \Gamma, (x : \mathsf{int}), \mathsf{even}(x) \vdash x + x : \mathsf{int}\ \{\lambda(x : \mathsf{int}).4 \text{ divides } x\}}{\Gamma \vdash \mathsf{let}\ (x\ : \mathsf{int}/\mathsf{even}(x)) = 2\ \mathsf{in}\ x + x : \mathsf{int}\ \{\lambda(x : \mathsf{int}).4 \text{ divides } x\}}$$

- In a surface language, we could encode assertions using **let**.

$$\mathsf{assert}\ F\ \mathsf{in}\ t \equiv \mathsf{let}\ (x\ : \mathsf{unit}/F) = ()\ \mathsf{in}\ t$$

Typing rule for case analysis

The operational semantics of pattern matching can be reflected at the logical level by inserting equalities (resp. inequalities) that correspond that the success (resp. failures) of the current branch (resp. previous branches).

Nil

$$\frac{\Gamma \vdash v : \tau \qquad \Gamma \models \mathsf{false}}{\Gamma \vdash \mathsf{case}\, v\, \mathsf{of}\, \emptyset : \tau'\ \{P\}}$$

Cons

$$\frac{\Gamma \vdash v : \tau \qquad \Gamma' \vdash p : \tau \qquad \Gamma, \Gamma', \lceil v \rceil = \lceil p \rceil \vdash t : \tau'\ \{P\} \qquad \Gamma, (\forall \Gamma'. \lceil v \rceil \neq \lceil p \rceil) \vdash \mathsf{case}\, v\, \mathsf{of}\, c : \tau'\ \{P\}}{\Gamma \vdash \mathsf{case}\, v\, \mathsf{of}\, p \Rightarrow t \mid c : \tau'\ \{P\}}$$

Example:

$$\frac{\Gamma \vdash \mathsf{o} : \mathsf{option(a)} \qquad (\mathsf{x} : \mathsf{a}) \vdash \mathsf{Some\ a\ (x)} : \mathsf{option(a)} \qquad \Gamma, \mathsf{o} = \mathsf{Some\ a\ (x)} \vdash \mathsf{x} : \mathsf{a}\ \{\lambda x.\mathsf{o} = \mathsf{Some\ a\ (x)}\} \qquad \Gamma, o \neq \mathsf{None}, (\forall(\mathsf{x} : \mathsf{a}).\mathsf{o} \neq \mathsf{Some\ a\ (x)}) \vdash \mathsf{case\ o\ of}\ \emptyset : \mathsf{option(a)}\ \{\lambda x.\mathsf{o} = \mathsf{Some\ a\ (x)}\}}{\Gamma, o \neq \mathsf{None} \vdash \mathsf{case\ o\ of\ Some\ a\ (x)} \Rightarrow \mathsf{x} \mid \emptyset : \mathsf{a}\ \{\lambda x.\mathsf{o} = \mathsf{Some\ a\ (x)}\}}$$

In this example, the environment of the last premise entails **false**.

Absurd

We can use an empty case analysis to encode dead code branch :

$$\mathsf{absurd} \equiv \mathsf{case}\,()\,\mathsf{of}\,\emptyset$$

Typing rule for function

The typing rule for function is natural: assume the precondition, check that the body of the function satisfies the postcondition:

Fun

$$\frac{\Gamma, (f : \tau_1 \to \tau_2), f = \lceil \textsf{fun}\ f \ldots \rceil, (x_1 : \tau_1), F_1 \vdash t : \tau_2\ \{\lambda(x_2 : \lceil \tau_2 \rceil).F_2\}}{\Gamma \vdash \textsf{fun}\ f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = t : \tau_1 \to \tau_2}$$

Example:

$$\frac{\begin{array}{c}\Gamma, (\mathsf{unSome} : \mathsf{option(a)} \to \mathsf{a}),\\ \mathsf{unSome} = (\lambda(\mathsf{o} : \mathsf{option(a)}).\mathsf{o} \neq \mathsf{None}, \lambda(\mathsf{o} : \mathsf{option(a)})(\mathsf{x} : \mathsf{a}).\mathsf{o} = \mathsf{Some\ a\ (x)})\\ (\mathsf{o} : \mathsf{option(a)}), \mathsf{o} \neq \mathsf{None}\\ \vdash t : \mathsf{a}\ \{\lambda(x : \mathsf{a}).\mathsf{o} = \mathsf{Some\ a\ (x)}\}\end{array}}{\Gamma \vdash \textsf{fun}\ \mathsf{unSome}(\mathsf{o} : \mathsf{option(a)}/\mathsf{o} \neq \mathsf{None}) : (\mathsf{x} : \mathsf{a}/\mathsf{o} = \mathsf{Some\ a\ (x)}) = t : \mathsf{option(a)} \to \mathsf{a}}$$

Inference of trivial postconditions

This propagation of postcondition gives what it was paid for: if no logic assertion is given, the proof system only ensures safety. In other words, when a placeholder for a logic assertion has not been filled, the formula $True$ is automatically used.

Nonetheless, a preprocessor can automatically infer trivial assertions on **let**:

$$\begin{aligned} \text{let x} = v &\rightsquigarrow \text{let x where } (\text{x} = \lceil v \rceil) = v \\ \text{let y} = f(x) &\rightsquigarrow \text{let y where post}(f)(x)(y) = f(x) \end{aligned}$$

Inferring strongest postconditions would be possible: yet, case analyses would generate an untractable combinatorial explosion due to disjunctions. A synthetic user-defined assertion is more useful.

Dealing with proof obligations

Proof obligations can be directly handled in Coq. Yet, the whole point to scale up the process is to use automatic efficient first-order provers. Proof obligations can be encoded into the polymorphic first-order logic using **defunctionalization** compilation technique. In a word, the idea is to reify the λ-terms of the higher-order logic as objects of the first-order logic and to axiomatize their reduction. This is an incomplete encoding because no new λ-terms can be "invented" by the prover. Yet, it does work well in practice [1]

Besides, the pre and post operators behave well through this encoding :

$$f = (\lambda x_1.F_1, \lambda x_1.\lambda x_2.F_2),$$

becomes

$$\forall x_1.(\mathsf{pre}(f, x_1) \Leftrightarrow F_1) \wedge \ \forall x_1.\forall x_2.(\mathsf{post}(f, x_1, x_2) \Leftrightarrow F_2)$$

If F_1 and F_2 are first-order formulæ, they are untouched by the encoding.

[1] A lot of optimization techniques can be applied to this encoding in order to ease automatic proof.

Experiment: the Set module of O'Caml

The code is directly imported from the standard library of Objective Caml. and its specification is based on an axiomatization of sets.

The verification-condition generator produces 804 proof obligations and 55 lemmas. They all were discharged by the Alt-Ergo prover, except 1 lemma, the positivity of the tree height which was shown by an induction in Coq.

Yann Régis-Gianas
yrg@irif.fr

Benchmarks

Function	P.O.	< 5s	< 30s	< 600s
is_empty	20	20	0	0
height?	5	5	0	0
mk_node	18	18	0	0
bal	179	143	23	13
add	39	33	6	0
join	82	64	6	12
min_elt	9	6	2	1
max_elt	11	7	3	1
remove_min_elt	22	16	4	3
merge	32	27	5	0
concat	27	25	2	0
split	75	55	9	11
remove	40	30	2	8
union	81	66	0	15
inter	44	34	1	9
diff	44	34	0	10
subset	48	42	0	6
fold	18	11	4	3
next	10	10	0	0
total	804	652	67	85

How to scale up this approach?

Automatic proof adaptation:

The previous figures show that automatic proving can be computationally expensive. Even if provability is unchanged by minor changes of the program, re-proving a slightly different version of a proof obligation is not immediate. This reduces the reactivity of the tool. How to reuse the proof trees of validated proof obligations to handle almost equivalent new proof obligations?

Imperative higher-order programs:

Imperative programming is not optional to implement a realistic programming language. Reasoning on imperative higher-order programs is an active field of research. How should we mix purely functional programs with impure ones? Some approaches are based on monads [**ynot08**], on functional translation of impure programs into pure ones [16], on hiding side-effects [17], ...

Communication between types and logic assertions:

On the one hand, type systems prove a lot of decidable properties that could be used by the logical side. On the other hand, maybe some undecidable conversion problems could be handled by the logical side and put back on the typing side.

Bibliography I


Siles, V. & Herbelin, H. Equality is typable in Semi-Full Pure Type Systems. in Proceedings, 25th Annual IEEE Symposium on Logic in Computer Science (LICS '10), Edinburgh, UK, 11-14 July 2010 (IEEE Computer Society Press, 2010) (cit. on p. 3).


Miquel, A. Classical modal realizability and side effects. http://www.pps.jussieu.fr/~miquel/publis/realizmod.pdf (cit. on pp. 3, 23).


Lebresne, S. in Automata, Languages and Programming (eds Aceto, L. et al.) 323–335 (Springer Berlin / Heidelberg, 2008) (cit. on p. 3).


Augustsson, L. Cayenne, a language with dependent types. SIGPLAN Not. **34,** 239–250. issn: 0362-1340. http://doi.acm.org/10.1145/291251.289451 (1 Sept. 1998) (cit. on pp. 3, 5).

Bibliography II


Chen, C. & Xi, H. Combining Programming with Theorem Proving. in Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (Tallinn, Estonia, Sept. 2005), 66–77 (cit. on pp. 3, 17).


Weirich, S., Voizard, A., de Amorim, P. H. A. & Eisenberg, R. A. A Specification for Dependent Types in Haskell. Proc. ACM Program. Lang. **1,** 31:1–31:29. issn: 2475-1421. http://doi.acm.org/10.1145/3110275 (Aug. 2017) (cit. on pp. 3, 6, 23).


Cardelli, L. A Polymorphic [lambda]-calculus with Type: Type. (Digital Systems Research Center, 1986) (cit. on p. 5).


Xi, H. & Pfenning, F. Dependent Types in Practical Programming. in Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages (San Antonio, Jan. 1999), 214–227 (cit. on pp. 6, 10).

Bibliography III


Freeman, T. & Pfenning, F. Refinement types for ML. **6** (ACM, 1991) (cit. on p. 6).


Fogarty, S., Pasalic, E., Siek, J. & Taha, W. Concoqtion: indexed types now!. in PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (ACM, Nice, France, 2007), 112–121. isbn: 978-1-59593-620-2. http://dx.doi.org/10.1145/1244381.1244400 (cit. on p. 17).


Strub, P.-Y. Coq Modulo Theory. English. 19th EACSL Annual Conference on Computer Science Logic. http://hal.archives-ouvertes.fr/inria-00497404/PDF/csl-2010.pdf (Aug. 2010) (cit. on p. 17).

Bibliography IV


Hoare, C. A. R. An axiomatic basis for computer programming. Commun. ACM **12**, 576–580. issn: 0001-0782. http://doi.acm.org/10.1145/363235.363259 (10 Oct. 1969) (cit. on p. 26).


Leino, K. R. M. Extended Static Checking: A Ten-Year Perspective. in Informatics (2001), 157–175 (cit. on p. 30).


Filliâtre, J.-C. & Marché, C. Multi-prover Verification of C Programs. in ICFEM (2004), 15–29 (cit. on p. 30).


Régis-Gianas, Y. & Pottier, F. A Hoare Logic for Call-by-Value Functional Programs. in Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC'08) **5133** (Springer, July 2008), 305–335. http://gallium.inria.fr/~fpottier/publis/regis-gianas-pottier-hoarefp.ps.gz (cit. on pp. 32, 33, 34, 35).

Bibliography V

Charguéraud, A. & Pottier, F. Functional Translation of a Calculus of Capabilities. in Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08) (Sept. 2008), 213–224. http://gallium.inria.fr/~fpottier/publis/chargueraud-pottier-capabilities.ps.gz (cit. on p. 56).

Pottier, F. Hiding local state in direct style: a higher-order anti-frame rule. in Twenty-Third Annual IEEE Symposium on Logic In Computer Science (LICS'08) (Pittsburgh, Pennsylvania, June 2008), 331–340 (cit. on p. 56).