Refinement Types for Liveness Properties in Denotational Semantics

Internship

Colin Riba
Plume team, LIP, ENS de Lyon
http://perso.ens-lyon.fr/colin.riba/
colin.riba@ens-lyon.fr

Context

Specifying infinite behaviours. Functional programming on infinite datatypes, such as non-wellfounded trees or streams (i.e. infinite-words), is by now well established thanks to declarative definitions and equational reasoning on high-level abstractions. We are interested in input-output properties of higher-order programs that handle such infinite data. Consider for instance the usual filter function on streams

\[
\text{filter : } (A \to \text{Bool}) \to \text{StrA} \to \text{StrA}
\]

\[
\text{filter } p (a :: x) = \text{if } (p a) \text{ then } a :: \text{filter } p x \text{ else } (\text{filter } p x)
\]

where \text{StrA} stands for the type of streams on \(A\). Assume \(p : A \to \text{Bool}\) is a total function that tests for a property \(\psi\). If \(x\) is a stream on \(A\), then \((\text{filter } p x)\) retains those elements of \(x\) which satisfy \(\psi\). The stream produced by \((\text{filter } p x)\) is thus only partially defined, unless \(x\) has infinitely many elements satisfying \(\psi\).

Logics like LTL, CTL or the modal \(\mu\)-calculus are widely used to formulate, on infinite objects, safety and liveness properties (see e.g. [HR07, BS07]). Safety properties state that some “bad” event will not occur, while liveness properties specify that “something good” will happen (see e.g. [BK08]). At the logical level, one typically uses temporal modalities like \(\Box (\text{always})\) or \(\Diamond (\text{eventually})\) to write properties of streams and specifications of programs over such data.

A possible liveness specification for filter asserts that \((\text{filter } p x)\) is a totally defined stream whenever \(x\) is a totally defined stream with infinitely many elements satisfying \(\psi\). We express this with the temporal modalities \(\Box\) and \(\Diamond\). Let \(A\) be finite, and assume given, for each \(a\) of type \(A\), a formula \(\delta_a\) which holds on \(b : A\) exactly when \(b\) equals \(a\).

Then \(\Box \forall_a \delta_a\) selects those streams on \(A\) which are totally defined. The formula \(\Box \Diamond \psi\) expresses that a stream has infinitely many elements satisfying \(\psi\). Our liveness specification for filter can thus be formulated as follows:

\[
\text{for all stream } x : \text{StrA}, \quad x \text{ satisfies } \Box \forall_a \delta_a \text{ and } \Box \Diamond \psi \implies (\text{filter } p x) \text{ satisfies } \Box \forall_a \delta_a
\]  \(1\)

Temporal refinement types. It is undecidable whether a given higher-order program satisfies a given input-output temporal property written with formulae of the modal \(\mu\)-calculus [KTU10]. Having a type system is a partial workaround to this obstacle, which moreover enables to reason compositionally on programs, by decomposing a specification to the various components of a program in order to prove its global specification.

\(1\) In the setting of e.g. [JR21], we would assume \(A = \sum_{i=1}^n 1\), with \(\delta_i\) representing the image of the \(i\)th injection.
Previous work [JR21] proposed a temporal refinement type system on top of the guarded \( \lambda \)-calculus, a higher-order programming language with guarded recursion [CBBGB16]. Guarded recursion is a simple device to control and reason about unfoldings of fixpoints, and which enforces the productivity of programs.\(^2\)

The productivity discipline of the guarded \( \lambda \)-calculus is however quite strict, and we aim at proposing a refinement type system for an higher-order programming language with general recursion able to represent potentially non-productive programs. We ultimately target a language such as FPC (see e.g. [Pie02]), that extends Plotkin’s seminal PCF [Plo77] with recursive types, and can be employed to model such infinite data structures.

An important example not handled in the guarded recursive setting of [JR21] is the non-productive (partial) function \( \text{filter} \) above. Typically, we want to express the aforementioned specification (1) with the following refinement types for \( \text{filter} p \).

\[
\text{filter} \ p \ : \ \{ \text{Str} A \mid \Box \bigvee_{a \in A} \delta_a \land \Box \Diamond \psi \} \longrightarrow \{ \text{Str} A \mid \Box \bigvee_{a \in A} \delta_a \}
\]  

(2)

**Stone duality.** An inherent difficulty with streams (and with non-wellfounded trees) is that (as opposed to e.g. integers) such objects are infinite by nature. It is thus not immediately clear what do we exactly mean by “the stream \( x \) satisfies \( \Box \Diamond \psi \)”. In our view, the above specification for \( \text{filter} \) should hold for any stream whatsoever, and not only for those definable in a given programming language.

The guarded recursive setting underlying [JR21] is known to represent faithfully the standard set-theoretic semantics of coinductive types [Møg14]. Hence, the expected meaning of a temporal logic on such types is clear.

On the other hand, in presence of general recursion, the natural semantics of types lives in denotational domains (in the sense of e.g. [AJ95, AC98, Str06]). This leads us to investigate temporal properties on infinite datatypes at the level of denotational semantics. Logics on top of domains are known since quite a long time. Our reference is the paradigm of “Domain Theory in Logical Form” (DTLF) [Abr91] (see also [Zha91]), which allows one to systematically generate a logic from a domain representing a type. These logics are actually obtained by Stone duality, which orchestrates a rich interplay between domain theory, logic and (point-free) topology. This area is presented under various perspectives in a number of sources. We refer to [Abr91, AC98] and (e.g.) [Joh82, Vic89, Vic07, Jun13, GL13, GvG23].\(^3\)

Preliminary works conducted in a recent internship [RS23] have shown that, in the case of domain-theoretic streams, the negation-free fragment of the temporal logic LTL has a nice representation by means of a logic called \textit{Geometric Logic}, which is at the core of the most general form of Stone duality. The formulae involved in the above liveness specification (1) are negation-free, and we have shown in [RS23] how their representation in geometric logic indeed conveys the good approximations to prove that the denotation of \( \text{filter} \) meets this specification.

To our knowledge, [RS23] is the first systematic duality-theoretic approach to liveness properties in a denotational setting.

**Objectives**

The broad objective is to devise a (syntactic) type system with refinement types allowing for expressing specifications like (1) as typing judgments like (2). The envisaged method is to adapt the guarded recursive system of [JR21] to the domain-theoretic setting of [RS23]. Depending on the tastes and skills of the student, there are several possible more specific targets.

First, while we know from [RS23] how to handle the specification (1) for \( \text{filter} \), we have yet no general reasoning principle for proving that a given (general) recursive function satisfies a given temporal (liveness) specification. Designing a sufficiently general proof rule be would a first objective.

The next step is to propose, in the case of streams, a refinement type system which incorporates a temporal logic together with a good representation of approximations of temporal properties. In this respect, most basic tools are available: refinement types à la [JR21], intermediate type systems based

\(^2\)On infinite datatypes such as streams, programs in general do not terminate. But the \textit{productive} ones compute a finite part of their output in finite time.

\(^3\)Some key ideas are also put at work in [CZ00].
on DTLF [Abr91, AC98], and the description of liveness properties at the denotational level [RS23]. We target refinement types for a \( \lambda \)-calculus with general recursion and a (simple) primitive type of streams.

A more ambitious task is to handle (alternation-free) modal \( \mu \)-properties on (finitely) polynomial types, thus targeting a system which as a whole would be based on FPC. Finitary polynomial types represent (finitely-branching) non-wellfounded trees, and usually involve sum types. But unfortunately, such sum types are not well-behaved in usual categories of domains. Fortunately, this limitation can be overcome by replacing the usual \( \lambda \)-calculus with the (more complex) setting of Call-By-Push-Value (CBPV) [Lev03, Lev22]. On the long run, it would be nice if this basis could extend to enriched models of CBPV, so as to handle further computational effects. Print and global store are particularly relevant, as an important trend in proving temporal properties considers programs generating streams of events.\(^4\)

A more prospective direction would be to extend our approach based on geometric logic to domain-theoretic semantics of linear logic [HJK00], for instance targeting trace-based systems like [NW03, Win04]. This may ultimately go as far as the categorical study of [BF06].

Last but not least, any competence and motivation for implementing a refinement type system (at least) for streams would be more than welcomed.

Depending on the orientation, this project may also involve collaborations with Guilhem Jaber and Kenji Maillard.

References


\(^4\)Major works in this line include e.g. [SSVH08, HC14, HL17, NUKT18, KT14, UST18, NUKT18, SU23].


