Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Exception tracking in an open world

Robert Harper

Carnegie Mellon University, Pittsburgh, PA 15217, United States

ARTICLE INFO

Article history:
Received 9 July 2017
Received in revised form 17 December 2017
Accepted 26 February 2018
Available online xxxx

Dedicated to Don Sannella on the occasion of his 60th birthday

Keywords:
Exception tracking
Type refinements
Open-world assumption
Dynamic classification
Program verification

ABSTRACT

A refinement is a predicate on the elements of a type that describes their execution behavior. Much work has gone into developing refinements in a *closed world*, in which the classes of values of a type are fixed *statically*, as in the case of the natural numbers with `zero` and `succ`. Relatively little work has gone into developing refinements in an *open world* in which new classes may be added *dynamically*. Here we examine the problem of *exception tracking*, a perennially problematic typing concept for programming languages, from the point of view of refinements in an open world. Exceptions are decomposed into separate control and data mechanisms, the latter motivating the need for open-world refinements. Exception tracking is thereby repositioned as a matter of program verification, rather than structural typing, integrating behavioral typing with theorem proving even in an open world. Some further applications of dynamic classification and open-world refinements are suggested.

1. Introduction

Don Sannella devoted much of his career to formal program development [1]. Much of his work emphasized equational reasoning, which is of fundamental importance. Don was also a contributor to the design of Standard ML [2], which included a notably powerful exception mechanism. Here we develop a logic for exception tracking in an ML-like setting using a relational program logic called type refinement. In the basic form considered here type refinements are properties of program behavior, but more general forms are binary relations specifying behavioral equality, a natural link to Don's work.

Structural type systems determine the grammar and dynamics of a language. Typing is expected to be decidable, and the dynamics is expected to be safe [3]. Behavioral type systems specify how a well-formed program behaves when executed. Typing cannot be expected to be decidable (except by being economical with the truth), but must be proved by a combination of human and mechanical reasoning. Structural typing is most closely related to proof theory [4], and the syntactic propositions-as-types principle [5]; behavioral typing is most closely related to realizability theory [6], and the semantic propositions-as-types principle [7,8].

Type refinements [9,10] were developed as a form of behavioral type system for functional programs. A type refinement is an inductively defined predicate over a type, making use of the action of type constructors on them and of logical concepts such as intersection, union, and entailment. Consider the following example in which a type of natural numbers (represented in unary) is defined, a function `inc` is defined over them, and two refinements (aka sorts), `even` and `odd`, are defined on `nat` that may be used to characterize the behavior of `inc`.

E-mail address: rwh@cs.cmu.edu.

https://doi.org/10.1016/j.tcs.2018.02.032
0304-3975/© 2018 Published by Elsevier B.V.

```
datatype nat = Zero | Succ of nat
datasort even = Zero | Succ of odd
and      odd = Succ of even
fun inc Zero = Succ Zero
  | inc (Succ n) = Succ  (inc n)
```

In the presence of such declarations, the function `inc` satisfies the refinement $(\texttt{even} \to \texttt{odd}) \wedge (\texttt{odd} \to \texttt{even})$ in the sense that it truly does carry even's to odd's and odd's to even's. The stated property is an example of a refinement that makes use of the predicate action of the function space, and of the intersection of two refinements. The assertion may be proved by a straightforward inductive argument that is readily mechanized, but in general such properties can encode uncomputable or unsolved problems, and hence cannot be expected to be checked by purely mechanical means. Yet much emphasis has been placed on inductive definitions of the *formal provability* of refinement satisfaction, $e \in_\tau \phi$, where $e : \tau$, and ϕ refines τ, rather than on building a framework in which one may prove that such judgments are *true*. For example, the judgment $e \in_{\texttt{nat}} \texttt{even}$ is true iff either e evaluates to $\texttt{zero}$ or to $\texttt{succ}(e')$ and $e' \in_{\texttt{nat}} \texttt{odd}$ true, and similarly for $e \in_{\texttt{nat}} \texttt{odd}$. Scaling up, $e \in_{\tau_1 \to \tau_2} \phi_1 \to \phi_2$ iff whenever $e_1 \in_{\tau_1} \phi_1$ holds, then $e(e_1) \in_{\tau_2} \phi_2$ holds as well.

Logical connectives, such as conjunction/truth and disjunction/falsehood are managed by *entailment* between refinements. The preorder $\phi \leq_\tau \psi$ between refinements of a type τ states that every value of type τ satisfying ϕ also satisfies ψ. This ordering is a lattice, which means that has finite meets and joins—the mentioned conjunctions and disjunctions needed for specification. Negation is a more delicate matter, raising the distinction between *closed-* and *open-world* type refinements. Refinements of a type such as `nat` are closed-world, because its *only* constructors are `Zero` and `Succ`. Consequently, we may reason that if a value of type `nat` is not `Zero`, then it must be `Succ` of something, and *vice-versa*. Negation in a closed world behaves classically in the sense that we have complete knowledge of what is and is not the case.

In an open world we cannot reason based solely on what we know now, but we must instead account for possible futures, much in the manner of intuitionistic logic. The *locus classicus* of such a situation arises with *exception tracking* of the kind found in languages such as CLU [11] in the '70's, FX [12] in the '80's, and Java [13] in the '90's. Exception names are analogous to constructors such as `Zero` and `Succ`, but there is no limit to the possible forms of exception that may be raised at a given program point, even ones that are not in scope. Higher-order features, such as functions and objects, give rise to such behavior. Moreover, in the interest of modularity and extensibility, exceptions are often dynamically generated, and hence cannot be named statically.

Negation plays an important role in exception-tracking. Standard accounts, such as those mentioned above, emphasize "positive" information—which exceptions may be raised—to the exclusion of "negative" information—which exceptions cannot be raised. But the negative information is just as important, if not more so. In a closed world the positive information determines, by implication, the negative information. But in an open world one must account for extensions, and the reasoning is not so simple.

Consider the following example:

```
let
  exception X
in
  raise X
end
```

The expression `raise X` raises the exception `X`, as does the entire expression. With positive exception tracking, we may assert within the scope of `X` that `raise X` does indeed raise `X`. But upon exiting the scope of the declaration, the exception `X` is still raised, but it cannot be named and hence cannot be specified in a positive tracking regimen. This renders positive information *unsound* in that an exception may be raised by an expression that is not (because it cannot) be specified in a type refinement. Note well that this discrepancy is not a matter of being conservative. It would be conservative to specify more exceptions than can actually be raised, but it is unsound to not specify an exception that can.

One consequence is that it is equally important to track both positive and negative information about exceptions. In a closed world a predicate and its negation are complementary, and so one determines the other. In an open world this is no longer the case. It becomes important, then, to track negative information—which exceptions cannot be raised—as much as positive—those that can be raised.

Consider the following example:

```
let
  exception X
in
  2+2
end
```

The body of the declaration cannot raise the exception `X`, and neither can the declaration as a whole. With the scope of `X` we may specify that `X` cannot be raised by an expression such as the addition in the above example. But this specification cannot be propagated beyond the scope of the declaration of `X`. *But neglecting to mention it affects only precision, not soundness.*