Type Refinements in an Open World (Extended Abstract)

Robert Harper\(^1\) and William Duff\(^1\)

\(^1\) Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15217, USA

### 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 \texttt{zero} and \texttt{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.

**Keywords and phrases** exception tracking, type refinements, open-world assumption, dynamic classification, program verification

### 1 Introduction

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 [8]. 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 [19], and the syntactic propositions-as-types principle [10]; behavioral typing is most closely related to realizability theory [13], and the semantic propositions-as-types principle [16, 4].

Type refinements [7, 6] 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 \texttt{inc} is defined over them, and two refinements (aka sorts), \texttt{even} and \texttt{odd}, are defined on \texttt{nat} that may be used to characterize the behavior of \texttt{inc}.

**Data Sort Refinement**

```ml
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)
```

© Robert Harper; licensed under Creative Commons License CC-BY
Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
In the presence of such declarations, the function \( \text{inc} \) satisfies the refinement \((\text{even} \rightarrow \text{odd}) \land (\text{odd} \rightarrow \text{even})\) 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 \( \phi \) refines \( \tau \), rather than on building a framework in which one may prove that such judgments are true. For example, the judgment \( e \in_{\text{nat}} \text{even} \) is true iff either \( e \) evaluates to \( \text{zero} \) or to \( \text{succ}(e') \) and \( e' \in_{\text{nat}} \text{odd} \) true, and similarly for \( e \in_{\text{nat}} \text{odd} \). Scaling up, \( e \in_{\tau_1 \rightarrow \tau_2} \phi_1 \rightarrow \phi_2 \) iff whenever \( e_1 \in_{\tau_1} \phi_1 \) holds, then \( e(e_1) \in_{\tau_2} \phi_2 \) holds as well. The entailment, \( \phi_1 \leq \phi_2 \), between refinements holds whenever \( e \in_{\phi_1} \), then \( e \in_{\phi_2} \), corresponding to the usual logical entailment, and \( \phi_1 \land \phi_2 \) and \( \phi_1 \lor \phi_2 \) are defined as the meet and join with respect to entailment. Negation is a more delicate matter involving the distinction between an open- and a closed world, analogous to the distinction between intuitionistic (open-world) and classical (closed-world) logic [11]. Familiar refinement systems are closed-world and hence may employ classical reasoning such as arguing that if a \( \text{nat} \) is not \( \text{Zero} \) then it must be a \( \text{Succ} \), and vice versa, precisely because the definition of \( \text{nat} \) is closed.

An awkward situation arises when data type and sort declarations are scoped, as in the following variation of the preceding example:

Scoped Data Sort Refinement

```plaintext
local
datatype nat = Zero | Succ of nat
datasort even = Zero | Succ of odd
and odd = Succ of even
in
  fun inc Zero = Succ Zero
  | inc (Suc n) = Succ (inc n)
end
```

The refinement of \( \text{inc} \) quoted above remains valid, but cannot be stated for the declaration of \( \text{inc} \), because \( \text{Zero} \) and \( \text{Succ} \) are out of scope. In a closed world where all classes are declared statically such embarassments may be avoided by suitably enlarging the scopes of the declarations, perhaps during elaboration [9]. But this move is not available in an open world, because the scopes of classes are determined dynamically, and not statically.

The locus classicus of the distinction is with exception tracking found in languages such as CLU [14] in the ’70’s, FX [15] in the ’80’s, and Java [12] in the ’90’s. These proposals have not worked out very well, particularly when higher-order programming is involved. The basic difficulty is that exception tracking is a behavioral, rather than a structural, property of a program; it is not a matter of grammar whether a program may incur an exception when executed. Moreover, in the interest of modularity and extensibility, it is important that exceptions be dynamically allocated; otherwise, one risks conflicts among components or instances of components, or must rely on whole-program, rather than separate, compilation. In the dynamic case it may not even be possible to name the exceptions that might be raised by an expression, let alone accurately track them.

Finally, previous accounts of exception tracking emphasize “positive” information—which exceptions may be raised—to the exclusion of “negative” exception tracking—which exceptions cannot be raised. But the negative information is just as important, if not more so, especially
when exceptions can be dynamic. First consider the following example:

Scoped Exceptions, I

```Scheme
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 regiment. This renders positive information unsound in the sense that an exception may be raised that cannot be stated to do so. At best one may consider positive tracking to mean “these exceptions may be raised, and so may some others”, from which it is impossible to deduce any information about what exceptions may not be raised.

This alone suggests that negative exception tracking—which exceptions cannot be raised—is at least as important as positive, if not more so, particularly when exceptions are dynamic. Moreover, negative information may validly be dropped on exit from the scope of an exception. Consider the following example:

Scoped Exceptions, II

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

The body cannot raise the exception `X`, and neither can the entire expression. This fact may be expressed within the scope of `X`, but cannot, as before, be propagated outside of its scope. But neglecting to mention it affects only accuracy, not soundness! It is not erroneous to fail to mention that an expression cannot raise `X`, merely imprecise when this is in fact the case.

## 2 Dynamic Classification

It is useful to decompose a conventional exception mechanism, such as is found in Java or SML, into two separate parts, a control mechanism, which effects a jump from the raiser to the handler, and a data mechanism, which transfers a value along with the transfer of control. The control aspect is standard; but what is to be the type of transferred values? In a closed world the type of exception values is essentially a (finite or infinite) sum, and we may reason knowing this fact. But it is more expressive to permit the exception type to be dynamically extensible, which is to say to work in an open world. ¹ Doing so avoids whole-program assumptions, and allows for multiple dynamic instances of program components without accidental identification of distinct exceptions.

¹ It is here that we differ with the account of Benton, et al. [3], who use a fixed, albeit countably infinite, set of exception classes.
The open world approach may be formulated by taking the exception value type to be the type \texttt{clsfd} of dynamically classified values.\footnote{This approach was introduced by Appel for Standard ML, and is closely related to Allen’s \cite{1} and Pitts and Stark’s \cite{18} name generation.} Values of type \texttt{clsfd} are labelled by a dynamically generated class that determines the type of the associated data. The introductory form \(a \cdot e\) attaches the class \(a\) with associated type \(\tau\) to the expression \(e\) of type \(\tau\); the associated type is maintained by a signature, \(\Sigma\), consisting of a finite set of declarations of the form \(a \sim \tau\), at most one for each \(a\). Classes are not variables, and are not values of any type; they are mere symbols. The eliminatory form is a one-sided pattern matching construct, \(\text{match } e \{ a \cdot x \mapsto e_1 \text{ ow } e_2 \}\), that tests whether a given value of type \texttt{clsfd} is tagged with the class \(a\); if so, the underlying value is bound to \(x\), and evaluation continues with \(e_1\); if not, evaluation proceeds with \(e_2\).\footnote{In contrast to Pitts and Stark the elimination form is not an equality test between two classes-qua-values, but rather a one-sided match of a class value against a given class.} A new class is introduced by \texttt{new } \(a \sim \tau \text{ in } e\) within \(e\), with no restriction on export of \(a\) outside of its static scope; this is what makes dynamic classification dynamic.

A selection of the structural typing rules for value-carrying exceptions and for dynamic classification are given in Figure 1.

The dynamics is relatively straightforward, combining standard structural rules for exceptions and for dynamic class generation. States have the form \(\nu \Sigma\{ e \}\), with a single-step transition relation between them. The judgment \(e \vdash_{\Sigma} v\) states that \(e\) is a value relative to \(\Sigma\), and the judgement \(e \uparrow_{\Sigma} v\) states that \(e\) raises an exception with value \(v\) relative to \(\Sigma\). The judgment \(\nu \Sigma\{ e \}\) \texttt{final} asserts that the given state is final (i.e., is a legitimate ending state). A selection of the rules of the dynamics is given in Figure 2.

### 3 Open-World Refinements

The simple closed-world satisfaction judgment, \(e \in_\tau \phi\), must be generalized to account for an open world and for the possibility of exceptions. \textit{Value satisfaction}, \(v \in_\tau \phi[\Sigma]\), asserts for \(v \downarrow_{\Sigma}\) that it satisfies the refinement \(\phi\) of the type \(\tau\). \textit{Expression satisfaction}, \(e \in_\tau \phi \uparrow \psi[\Sigma]\), states that if \(\nu \Sigma\{ e \}\) is executed to completion, then it either results in a value satisfying \(\phi\) or raises an exception value satisfying \(\psi\).

Because function bodies can raise exceptions, function refinements have the form \(\phi \rightarrow \psi_1 \uparrow \psi_2\), where \(\phi\) governs the domain, and the \(\psi_i\) govern the normal and exceptional return value. We consider two refinements of \texttt{clsfd}, written \(a \cdot\) and \(\bar{a} \cdot\), which specify whether the
For the sake of simplicity, recursive types are not considered, but standard methods such as step-collections of relevant facts, stated in rule form for perspicuity. Any such inductive definition as an inductive definition of a collection of formally derivable judgments, but rather as a rules are given in Figure 4. It is important to understand that these rules are be valid iff the truth of its premises implies the truth of its conclusion. Some valid refinement refinements \( \Phi \) satisfaction to hold for all substitution instances of \( \phi \) and \( \tau \). A sample of the clauses defining the refinement structure, with intersections as meets and unions as joins, and top and bottom elements that every value satisfying \( \Sigma \) class of a value is or is not \( a \). Entailment, written \( \phi \leq \Sigma \) \( \phi' \), carries over with the meaning that every value satisfying \( \phi \) also satisfies \( \phi' \), relative to \( \Sigma \). Entailment enjoys a lattice structure, with intersections as meets and unions as joins, and top and bottom elements given by truth and falsity, respectively. A sample of the clauses defining the refinement judgments are given in Figure 3. The value and computation judgments are defined mutually recursively by induction on the structure of types.\(^4\)

Suppose that \( \Gamma \vdash \Sigma \ e : \tau \) is a well-typed expression, and that \( \Phi \) refines \( \Gamma \) variable-wise, and \( \phi \) refines \( \tau \), both relative to \( \Sigma \). The judgment \( \Phi \vdash \Sigma \ e : \tau \) extends expression satisfaction to hold for all substitution instances of \( e \) by values satisfying, variable-wise, the refinements \( \Phi \). An inference rule with such judgments as premises and conclusion is said to be valid iff the truth of its premises implies the truth of its conclusion. Some valid refinement rules are given in Figure 4. It is important to understand that these rules are not intended as an inductive definition of a collection of formally derivable judgments, but rather as a collection of relevant facts, stated in rule form for perspicuity. Any such inductive definition

\(^4\) For the sake of simplicity, recursive types are not considered, but standard methods such as step-indexing [2] and Pitts’s ST-closure [17] seem to be applicable.
would be woefully incomplete with respect to validity as just defined, but might be useful for automation.

Rule 1 states that the refinement of the normal and exceptional returns may be weakened. This is typically required to ensure conformity at a join point, where information loss can be unavoidable. Rules 2 govern the classification of values in the evident manner, relying on being able to test disequality of classes. Rules 3, 4, and 5 govern class matching, according to whether the class is known to be the matching class, \( a \), or known not to be \( a \), or not known either way. Notice that in Rules 3 and 4 one of the two clauses is disregarded because the branch outcome is known.\(^5\) Rules 6 state that the raised value raised by a \( \texttt{raise} \) satisfies the refinement of the raised value, and does not return a normal value, and propagates what is known about the raised value of \( e \) to the otherwise clause of a \( \texttt{try} \), and otherwise agrees

\(^5\) The matching construct could be generalized so as to rebind the matched expression to a bound variable of each branch, so that the fact that the class of \( e \) is or isn’t \( a \) is propagated appropriately.
with the normal value.

Finally, Rule 7 states that any property of the body of a class declaration holds of the whole declaration, provided that it does not mention the newly declared class $a$. Generally speaking, the restrictions on $\phi$ and $\psi$ are achieved by weakening, and it is here that the emphasis on negative information comes into play. Specifically, if $\psi$ were a conjunction of negated class refinements, $\overline{a_1} \land \cdots \land \overline{a_n}$, then any of the conjuncts could be dropped by applying the entailment

$$\overline{a_1} \land \cdots \land \overline{a_n} \leq \overline{a_1} \land \cdots \land \overline{a_k} \quad (1 \leq k \leq n).$$

If $\psi$ were instead the more familiar disjunction of positive class refinements, $a_1 \lor \cdots \lor a_n$, then no disjunct may be soundly dropped, for to do so would amount to a use of the invalid entailment

$$a_1 \lor \cdots \lor a_n \not\leq a_1 \lor \cdots \lor a_k \quad (1 \leq k < n).$$

In a closed world one may consider that exception names never go out of scope, escaping the difficulty. But in an open world classes cannot always be named, raising serious obstacles for positive exception tracking.

## 4 Conclusion

Structural typing is a matter of grammar; behavioral typing is a matter of verification. Whereas conformance with grammar can be expected to be machine-checkable, conformance with a behavioral specification cannot. Separating the two concepts resolves long-standing issues, such as are encountered with exception tracking in languages like Java. In ongoing work we are developing tactics for proving behavioral properties of programs in an open world using the Coq prover [5]. A significant challenge is to extend the Coq type theory, as Allen has done for NuPRL [1], with dynamic class generation. In future work we plan to apply these methods to exception tracking, and to other applications of open-world refinements, such as in language-based security.

### Acknowledgements

The authors are grateful to Rowan Davies for his advice and comments, and to Arbob Ahmad, Carlo Angiuli, Karl Crary, Favonia, and Dan Licata for their comments and suggestions.

### References