Runtime checks as nominal types

Paola Giannini[1], Marco Servetto[2], and Elena Zucca[3]

[1] Università del Piemonte Orientale, Italy
giannini@di.unipmn.it
[2] Victoria University of Wellington, New Zealand
marco.servetto@ecs.vuw.ac.nz
[3] Università di Genova, Italy
elena.zucca@unige.it

Abstract. We propose a language design where types can be enriched by *tags* corresponding to predicates written by the programmer. For instance, `int&positive` is a type, where `positive` is a user-defined boolean function on integers. Expressions of type `int&positive` are obtained by an explicit *check* construct, analogous to cast, e.g., `(positive) 2`. In this way, the fact that the value of an expression is guaranteed to succeed a runtime check is a *static* property which can be controlled by the type system. We formalize our proposal as an extension of the simply-typed lambda calculus, and prove, besides soundness, the fact that expressions of tagged types reduce to values which satisfy the corresponding predicates.

1 Introduction

It is very common in programming that an application needs to handle only values which satisfy properties which can be checked through user-defined code. As a very simple example, integer numbers which are odd, or positive, or prime. In nominally typed languages, it is possible to declare specialized types which enforce/encode such invariants. This is obtained by performing runtime checks on the arguments of the constructor of the new type. There are two variations of this technique, the first is *wrapping* the original type, and the second is *extending* it. We describe them, by examples, using Java syntax.

The first solution is illustrated by the example of odd numbers:

```
class Odd{
  public final int inner;
  public Odd(int inner){
    if (inner%2==0){//not odd
      throw new Error(...); }
```

V. Biló, A. Caruso (Eds.): ICTCS 2016, Proceedings of the 17th Italian Conference on Theoretical Computer Science, 73100 Lecce, Italy, September 7–9 2016, pp. 75–87 published in CEUR Workshop Proceedins Vol-1720 at http://ceur-ws.org/Vol-1720

```
    this.inner=inner;
  }
}
```

Here we encode by wrapping, in such a way that all instances of `Odd` will always contain an odd number. In particular, every method taking as input an `Odd` argument `myOdd` can rely on the (static) guarantee that `myOdd.inner` is an odd number.

The second solution is illustrated by an example of points with positive coordinates:

```
class Point{int x; int y;...}
class PositivePoint extends Point{
  public PositivePoint(int x, int y){
    super(x,y);//checks parent invariant
    if (x<0 || y<0){//not positive
      throw new Error(...); }
  }
}
```

Here we encode by extending the original type (class), calling the super constructor and then checking for more properties. Again, every method taking in input a `PositivePoint` may assume that the coordinates are positive numbers.

Both patterns could be verified by tools like Spec# [3] or Viper [9].[4]

Comparing the two solutions, we can say that:

– Wrapping can be always applied, while extending cannot be applied on primitive types and final classes.
– While extending implies subtyping, a wrapped value has not a subtype of the original type, thus forcing extra boilerplate code in the points of usage to access the inner value.
– On the other hand, wrapping can be applied to existing values, while extending requires the creation of a new value.
– Anyway, both techniques require some amount of boilerplate code.

We propose an alternative approach, whose goal is to synthesize the core concept behind these techniques: a user-defined predicate is checked on a value, and, in case of success, the value gets a more specific type, that keeps track of the success of the runtime test. For instance, we can declare two functions checking whether an integer is odd or positive, respectively:

```
bool odd(int i){return i%2!=0;}
bool positive(int i){return i>=0;}
```

[4] Viper requires to encode the invariant as constructor postcondition, and to repeat it in the method precondition, while Spec# needs the assertion to be explicitly stated in the class contract, and allows strengthening of superclass invariants (as needed in `PositivePoint`).

Then, we can convert an integer to an odd, or to a positive, by a *check* construct, which has a cast-inspired syntax, and indeed can be seen as a generalization of cast:[5]

```
(odd)35//ok
(positive)-23 //runtime error
(positive)(odd)79 //checks chain
```

The expressions above have, respectively, *tagged* types: `int&odd`, `int&positive`, and `int&odd&positive`. Tagged types are *nominal*, since tags are predicate names, and behave as intuitively expected: tag sequences can be seen as sets (order and repetition are immaterial), and subtyping corresponds to set inclusion. They can be used, e.g., as types of local variables:

```
int&positive&odd myInt=(positive)(odd)79
int&positive anotherInt=myInt //safe, thanks to subtyping
```

or as types of function parameters/result:

```
int&positive factorial(int&positive n){
  if (n==0){return 1;}
  return n * factorial((positive) n-1); }
```

Note that the function does not need to check whether the argument is positive (to avoid non termination), since this is ensured by the parameter type. However, we need the check to perform the recursive call.

The paper is organized as follows. We formalize our proposal and prove its expected properties in Sect.2. In Sect.3 we discuss related work, and present our conclusions and further work in Sect.4.

2 The calculus

In this section we introduce syntax, operational semantics, and type system of our calculus. Moreover, we prove expected properties.

Syntax We formalize our proposal as an extension of the simply-typed (call-by-value) lambda calculus. Syntax is given in Fig.1. We assume an infinite set of *predicate names* P.

Expressions include expressions of the lambda calculus (variable, abstraction and application) and those defined with operators of primitive types, e.g., the two boolean constants, and the infinite set of integer constants, ranged over by n. For simplicity we omit other standard additional constructs, such as conditional, let-in and recursion. In addition, there are two novel expressions:

– `(`P`)`e is a *check expression*, corresponding to a runtime check that (the value of) expression e satisfies predicate P. If the runtime check succeeds, then the result of the check expression will be the value of e tagged by P. In case of failure, the result will be an error tagged by P.

[5] That is, a standard cast expression `(C)e` can be seen as a particular case of check expression where the predicate is `instanceof C`.

$p ::= P\texttt{=}\lambda x : T.e$	predicate definition
$e ::=$	expression
$\quad x \mid \lambda x : T.e \mid e_1\ e_2$	
$\quad \mid \texttt{true} \mid \texttt{false} \mid n \mid \ldots$	
$\quad \mid \texttt{(}P\texttt{)}\,e$	check expression
$\quad \mid v\&P[e]$	checking expression
$v ::=$	base value
$\quad \texttt{true} \mid \texttt{false} \mid n \mid \ldots$	primitive value
$\quad \mid v\&P$	tagged value
$V ::= \lambda x : T.e \mid v$	value
$\mathcal{E} ::= [\,] \mid \mathcal{E}\ e \mid V\,\mathcal{E}$	evaluation context
$\quad \texttt{(}P\texttt{)}\mathcal{E} \mid v\&P[\mathcal{E}]$	
$t ::=$	base type
$\quad \texttt{bool} \mid \texttt{int} \mid \ldots$	primitive type
$\quad \mid t\&P$	tagged type
$T ::= T_1 \to T_2 \mid t$	type
$\Gamma ::= x_1{:}T_1 \ldots x_n{:}T_n$	type context
$\Delta ::= P_1{:}T_1 \ldots P_n{:}T_n$	predicate type context

Fig. 1: Syntax

– $v\&P[e]$ is a *checking expression*, a runtime expression (that cannot be written by the programmer) which denotes an intermediate step in the runtime check that value v satisfies predicate P. That is, if e is neither `true` nor `false`, then we still have to evaluate e to complete the check. The checking expressions $v\&P[\texttt{true}]$ and $v\&P[\texttt{false}]$, instead, denote the final result of the check, being success and failure, respectively.

Accordingly with the intuition given above, we will use the following two shortcuts:

– $v\&P[\texttt{true}]$ will be abbreviated by $v\&P$. This is a *tagged value*, denoting the result of a successful check of the predicate, hence we can rely on the fact that $P\ v$ holds.
– $v\&P[\texttt{false}]$ will be abbreviated by $\texttt{error}\ P$. This is a dynamic error, tagged by P to denote that there has been a failure in checking P.

Values are abstractions, or values of primitive types, or tagged values as explained above. Note that we take a stratified approach, where only non-functional values can be tagged. Tagged values are identified modulo order and repetitions of tags, e.g., 1&`positive`&`odd` and 1&`odd`&`positive` are the same value.

Operational Semantics The reduction relation $e \to e'$ assumes a given *predicate table* $PT = p_1 \ldots p_n$, which is a sequence of *predicate definitions* which associate to a predicate name a lambda expression (expected to be a predicate, that is, to return a boolean value). We assume, as usual, that order of definitions is immaterial, and there are no multiple definitions for the same predicate name. That is, the predicate table can be seen as a (partial) map from predicate names to predicates.

Reduction rules are given in Fig.2.

$$\text{(Ctx)}\ \frac{e \rightarrow e'}{\mathcal{E}[e] \rightarrow \mathcal{E}[e']} \qquad \text{(Ctx-Err)}\ \mathcal{E}[\texttt{error}\ P] \rightarrow \texttt{error}\ P$$

$$\text{(App)}\ (\lambda x : T.e)\ V \rightarrow e[V/x]$$

$$\text{(Check)}\ (P)\,v \rightarrow v\&P[(\lambda x : T.e)\ v] \quad \begin{array}{l} PT(P) = \lambda x : T.e \\ v \text{ not of shape } v'\&P \end{array}$$

$$\text{(No-Check)}\ (P)\,v\&P \rightarrow v\&P$$

Fig. 2: Reduction rules

The first two rules are standard contextual closure and error propagation rules. Evaluation contexts, given in Fig.1, are as expected. The third rule is (call-by-value) application rule. We denote by $e[e'/x]$ the standard capture-avoiding substitution of occurrences of x in e by e'.

The novel rule (Check) models starting the runtime check that predicate named P holds on value v, by triggering the evaluation of the corresponding predicate application. This rule is applied when v is not tagged by P yet. Otherwise, this means that v is a value obtained through a previous check of predicate named P, hence it is useless to perform the check again, and the value is directly returned, as shown in rule (No-Check). Note that an implementation could keep the tags and follow the same strategy, avoiding useless checks, or be based on type erasure, but in this case checks should be repeated.

Type System Following the stratified approach mentioned above, there are two forms of types:

- non-functional types, called *base types*, which are primitive types possibly tagged with a sequence of predicate names
- functional types, which cannot be tagged.

As for tagged values, we assume that tagged types are identified modulo order and repetitions of tags, e.g., `int`&`positive`&`odd` and `int`&`odd`&`positive` are the same type.

The subtyping relation is given in Fig.3. Rule (Sub-Tag) expresses the expected property that adding tags we get more specific types, other rules are standard.

The typing judgment, $\Gamma \vdash e : T$, says that expression e has type T under the *type context* Γ, where type contexts, Γ, are sequences of assignments of types to

$$\text{(Sub-Tag)}\quad t\&P \leq t \qquad\qquad \text{(Sub-Fun)}\quad \frac{T_1' \leq T_1 \quad T_2 \leq T_2'}{T_1 \rightarrow T_2 \leq T_1' \rightarrow T_2'}$$

$$\text{(Sub-Tr)}\quad \frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3} \qquad \text{(Sub-Refl)}\quad T \leq T$$

Fig. 3: Subtyping rules

variables, see Fig.1. In Fig.4 we give the rules for deriving the typing judgment. We assume a given *predicate type context* Δ, which is a sequence of assignments of types to predicate names, see Fig.1.

$$\text{(T-Var)}\quad \frac{}{\Gamma \vdash x : T}\ \Gamma(x) = T \qquad \text{(T-Abs)}\quad \frac{\Gamma, x{:}T \vdash e : T'}{\Gamma \vdash \lambda x : T.e : T \rightarrow T'}$$

$$\text{(T-App)}\quad \frac{\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T'}\ T_2 \leq T$$

$$\text{(T-Check)}\quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{(}P\texttt{)}\,e : t\&P}\ \begin{array}{l}\Delta(P) = t' \rightarrow \texttt{bool}\\ t \leq t'\end{array}$$

$$\text{(T-Checking)}\quad \frac{\Gamma \vdash v : t \quad \Gamma \vdash e : \texttt{bool}}{\Gamma \vdash v\&P[e] : t\&P}\ \begin{array}{l}\Delta(P) = t' \rightarrow \texttt{bool}\\ t \leq t'\end{array}$$

$$\text{(T-Error)}\quad \frac{\Gamma \vdash v : t}{\Gamma \vdash v\&P[\texttt{false}] : T}\ \begin{array}{l}\Delta(P) = t' \rightarrow \texttt{bool}\\ t \leq t'\end{array}$$

Fig. 4: Typing rules

The first three rules of Fig.4 are standard rules of the simply-typed lambda calculus. We omit obvious rules for boolean and integer constants.

In rule (T-Check), a check expression has the type of the argument tagged by the predicate name. The argument type t must be a subtype of the parameter type of the predicate. Note that the explicit subtyping condition, rather than an implicit subsumption rule, is necessary to assign to the check expression type $t\&P$, which is possibly more specific than $t'\&P$. For instance, the predicate `positive` requires an `int`, the argument could be an `int&odd`, and we want to get an `int&odd&positive`.

Rule (T-Checking) enforces the restriction that, in a checking expression of type $t\&P$, the value v must have type t, and the expression e must be of type `bool`, so that, if it converges to a value, then the value is either `true` or `false`. As for rule (T-Check), the type of the value must be a subtype of the parameter type of the predicate.

Finally, rule (T-Error) states that an error has any type.[6] Since we encode errors as checking expressions where the expression is **`false`**[7], we also require for uniformity the same conditions on v of the two previous rules.

Results To prove soundness, expressed as usual by subject reduction and progress theorems, we assume that the predicates in the predicate table be well-typed w.r.t. the predicate type context, that is, for all $P \in \mathtt{dom}(PT)$, $\emptyset \vdash PT(P) : \Delta(P)$. The proof of subject reduction relies on the (standard) substitution and context lemmas, that follow.

Lemma 1 (Substitution). *If $\Gamma, x{:}T' \vdash e : T$, and $\Gamma \vdash e' : T'$, then $\Gamma \vdash e[e'/x] : T$.*

Lemma 2 (Context). *Let $\Gamma \vdash \mathcal{E}[e] : T$, then $\Gamma \vdash e : T'$ for some T', and if $\Gamma \vdash e' : T''$, with $T'' \leq T'$, then $\Gamma \vdash \mathcal{E}[e'] : T$, for all e'.*

Theorem 1 (Subject reduction). *Let e be such that, for some Γ and T, we have that $\Gamma \vdash e : T$. If $e \rightarrow e'$, then $\Gamma \vdash e' : T'$ for some T' such that $T' \leq T$.*

Proof. By induction on the rule used for $e \rightarrow e'$.

If the rule applied is (Ctx), then we have the thesis by induction hypothesis, using Lemma 2.

If the rule applied is (Ctx-Err), since **`error`** P is abbreviation for $v\&P[\mathtt{false}]$, from (T-error) of Fig.4 we have that $\Gamma \vdash v\&P[\mathtt{false}] : T$. This proves the thesis.

If the rule applied is (App), then $e = (\lambda x : T_1.e_1)\ V$, $e' = e_1[V/x]$. The proof is standard by using typing rules (T-App), (T-Abs), and Lemma 1.

If the rule applied is (Check), then

1. $e = \mathtt{(}P\mathtt{)}\,v$,
2. $e' = v\&P[(\lambda x : t'.e_p)\ v]$, where $PT(P) = \lambda x : t'.e_p$.

From $\Gamma \vdash \mathtt{(}P\mathtt{)}\,v : T$, and typing rule (T-Check) for some t and t' we have that: $T = t\&P$, $\Gamma \vdash v : t$, $\Delta(P) = t' \rightarrow \mathtt{bool}$, and $t \leq t'$. Moreover, we know that $\emptyset \vdash PT(P) : t' \rightarrow \mathtt{bool}$. Therefore, also $\Gamma \vdash PT(P) : t' \rightarrow \mathtt{bool}$. Applying rule (T-app) of Fig.4, we get that $\Gamma \vdash (\lambda x : t'.e_p)\ V : \mathtt{bool}$. Therefore, from $\Gamma \vdash v : t$, $\Gamma \vdash (\lambda x : t'.e_p)\ V : \mathtt{bool}$, $\Delta(P) = t' \rightarrow \mathtt{bool}$, and $t \leq t'$, applying rule (T-checking) of Fig.4, we derive $\Gamma \vdash e' : T$.

If the rule applied is (No-Check), then

1. $e = \mathtt{(}P\mathtt{)}\,v\&P[\mathtt{true}]$, and
2. $e' = v\&P[\mathtt{true}]$.

[6] Recall that this is a standard typing rule in calculi with dynamic errors, needed to ensure that error propagation, see rule (Ctx-Err), preserves type.

[7] We could have, alternatively, one more reduction step into an additional error expression, but we prefer this more succinct formulation.

From $\Gamma \vdash e : T$, and typing rule (T-Check) for some t and t' and we have that: $T = t\&P$, $\Gamma \vdash v\&P[\texttt{true}] : t$, $\Delta(P) = t' \rightarrow \texttt{bool}$, and $t \leq t'$. From $\Gamma \vdash v\&P[\texttt{true}] : t$, and typing rule (T-Checking) we have that t is already of shape $t''\&P$, for some t''. Therefore, $t = T$, and $\Gamma \vdash e' : t$. □

The proof of progress relies on the canonical forms lemma, which characterizes the shape of values of specific types.

Lemma 3 (Canonical forms).

1. *If* $\vdash V : T_1 \rightarrow T_2$, *then* $V = \lambda x : T_1.e$.
2. *If* $\vdash V : \texttt{bool}$, *then either* $V = \texttt{false}$, *or* $V = \texttt{true}$.
3. *If* $\vdash V : \texttt{int}$, *then* $V = n$.
4. *If* $\vdash V : t\&P$, *then* $V = v\&P$, *i.e.,* $V = v\&P[\texttt{true}]$.

Theorem 2 (Progress). *Let* e *be such that, for some* T, *we have that* $\emptyset \vdash e : T$. *Then, either* $e = V$, *for some* V, *or* $e = \texttt{error}\ P$, *or* $e \rightarrow e'$ *for some* e'.

Proof. If $e \neq V$, for some V, and $e \neq \texttt{error}\ P$, then either $e = e_1\, e_2$, or $e = (P)\, e_1$, or $e = v\&P[e_1]$. We consider the last two cases.

If $e = (P)\, e_1$, and e_1 is not a value or error, then by induction hypothesis $e_1 \rightarrow e_2$. Therefore, $(P)\, e_1 \rightarrow (P)\, e_2$ with (Ctx) and $\mathcal{E} = (P)[\ ]$.
If $e_1 = \texttt{error}\ P$, then $e \rightarrow \texttt{error}\ P$ with (Ctx-Err) and $\mathcal{E} = (P)[\ ]$.
If $e_1 = V$, for some V, from $\emptyset \vdash (P)\, V : T$, and typing rule (T-Check), we have that, for some t and t': $T = t\&P$, $\emptyset \vdash V : t$, $\Delta(P) = t' \rightarrow \texttt{bool}$, and $t \leq t'$.
There are two cases for type t: either $t = t'\&P$ for some t', that is, the type is already tagged with P, or not.
If $t = t'\&P$ for some t', then, from Lemma 3.4, $V = v\&P$, and rule (no-check) is applicable. So $e \rightarrow v\&P$.
Otherwise rule (check) is applicable, and $e \rightarrow V\&P[(\lambda x : t'.e_p)\ V]$.

If $e = v\&P[e_1]$, and e_1 is not a value, as in the previous case we can apply either rule (Ctx) or (Ctx-Err).
If $e_1 = V$, for some V, then, from $\emptyset \vdash v\&P[e_1] : T$, and typing rule (T-Checking), we have that $\emptyset \vdash e_1 : \texttt{bool}$. Therefore, from Lemma 3.2, $e_1 = \texttt{true}$ or $e_1 = \texttt{false}$. If $e_1 = \texttt{true}$, then e is the value $v\&P$, and if $e_1 = \texttt{false}$, then $e = \texttt{error}\ P$. □

In addition to soundness, we expect the following key property to hold: if an expression of a tagged type $t\&P$ reduces to a value, then this value satisfies predicate P. This is implied by subject reduction, plus the fact that a value v of a tagged type $t\&P$ satisfies predicate P. The latter property is true "by construction" in our calculus, provided that we consider as values only the subset of syntactic values which can be actually obtained by reduction. (Recall that checking expressions cannot be written by the programmer.) This is formalized by the notion of *admissible expression* below.

Definition 1. *The expression* e *is* admissible *if, for all subexpressions* $v\&P[e']$ *of* e, *we have* $(\lambda x : T.e_p)\ v \rightarrow^\star e'$, *where* $PT(P) = \lambda x : T.e_p$.

Note that expressions written by the programmer are always admissible since they do not have subexpressions which are checking expressions.

Lemma 4. *If e is admissible, and $e \to e'$, then e' is admissible.*

Proof. By induction on the reduction rules. The interesting case is:
(Check) $\texttt{(}P\texttt{)}\, v \to v\&P[(\lambda x : T.e)\ v]\ PT(P) = \lambda x : T.e$
In this case, the thesis holds since $(\lambda x : T.e)\ v$ reduces to itself in zero steps. □

Theorem 3. *Let e be an admissible expression such that, for some Γ and T, we have that $\Gamma \vdash e : T$. If $e \to^\star v\&P$, then $(\lambda x : T.e_p)\ v \to^\star$ `true`, where $PT(P) = \lambda x : T.e_p$.*

Proof. By subject reduction (Theorem 1) and Lemma 4. □

3 Related work

Constrained types and type invariants The work most closely related to ours are likely *constrained types* of X10 [10]. A constrained type `C{c}` consists of a class or interface `C` and a constraint `c` on the immutable state of `C` and in-scope final variables. The system is parametric on the underlying constraint system: the compiler implements simple equality-based constraints but, in addition, supports extension with new constraint systems using compiler plugins. For instance, the constraint solver automatically detects subtyping in cases such as `Int[self>5]` $\leq$ `Int[self>0]`. Their main goal is to support static verification as much as possible, by allowing in constraints a very restricted subset of the language.[8] On the other hand, in our approach the programmer can express *any* property in the predicate language, which is Turing complete being the language itself. This could be coupled with static verification for conditions expressed in a limited sub-language, but we do not require all the predicates to be in such category.

Whiley [11] offers *type invariants*, conceptually similar to constrained types but, as in our approach, they use the full language in predicates. Whiley is designed from scratch, with the aim of permitting both dynamic (runtime) verification and static one. As X10, Whiley attempts at automatically inferring predicate subtyping. In a personal communication with David Pearce (Whiley project leader) he agreed that static verification of predicates and subtyping inference are orthogonal with respect to the main idea of checking a property and then propagating such knowledge, and that a core model of such idea would be very valuable.

Another important difference is that, in both X10 and Whiley, interpretation of predicates is *structural*, that is, the meaning of a predicate is its definition. Our interpretation is *nominal* instead, that is, , the meaning of a predicate is its name, and the current definition is just an implementation of such concept. Under our lens, predicate subtyping must be defined explicitly by the programmer, if needed, see Sect.4.

[8] The user is able to escape the confines of static type-checking using dynamic casts, as in our approach.

Pre/post conditions In the style of runtime verification, a method can have *preconditions* on its arguments that are dynamically checked when the method is invoked. If the preconditions do not hold, the caller is blamed. Then, before producing the result, the *postcondition* is verified. If the postcondition does not hold, the method implementation is blamed, see, e.g., [8].

In our approach, a method can encode preconditions as types of its parameters, guaranteeing that the check is performed at the client side.[9] Then, the method may cast the result to a more specific type, as a way to check the postcondition. An advantage of encoding a precondition as a type is that in this way the information that a value satisfies the precondition can be propagated to internal calls with the same precondition, rather than be checked again.

Runtime certification Our work can be seen as a (circular) variation over *runtime certification* as in Athena [2]. In a nutshell, a program is runtime certified if, when results are produced, they are guaranteed to respect certain properties. Runtime certification is usually proposed together with a specific proof language, different from the computation language. Our approach instead uses a single language, and the correctness of a value is defined as the satisfaction of certain predicates written in the language itself.

Refinement types and blame calculi *Refinement types*, introduced in [4], are a system of subtypes for a polymorphically typed language like ML, which are used to specify properties of user-defined data types. The properties are expressed as union and intersection of types derived from user defined-types, and are statically verified by a decidable type inference algorithm. In [7] are defined two *hybrid calculi*, λ^C and λ^H, whose types include refinement types describing subsets of base types satisfying predicates. Predicates are, as in our calculus, expressions of the language evaluating to boolean values. Expressions include a cast, like our check, which involves checking dynamically that a predicate holds. As for X10 and Whiley, the interpretation of predicates is structural.

The *blame calculus*, see [12], provides a uniform view of static and dynamic type checking. Programmers may add casts to evolve statically typed code into code including refinement types (as in [7], predicates are structural). The blame calculus is more general than our calculus, since refinement is not limited to base types, and, in [1], it is extended to a polymorphically typed lambda calculus. The type system is tailored to detect where the cause of the violation comes from, and, in particular, it is proved that it may not come from statically typed code. We claim that, even though our framework is more limited, since we do not allow properties of functional types, its simplicity, and the nominal approach to refinement, make it easier to implement and integrate in existing languages, see next section.

[9] Preconditions involving more than one parameter could be encoded by using tuple types.

4 Conclusion and future work

We have proposed and formalized a *lightweight* approach for handling values which are required to satisfy a property.

Compared to works discussed in the previous section, which generally rely on separate assertion language, structural interpretation of predicates, and (some) static verification, our design is less powerful, but easier to integrate and implement in existing languages.

In particular, we adopt a meta-circular approach, where the property is implemented by the programmer in the language itself. Note that, in this way, checking that a value has a given property, say P, could lead to divergence. However, the fact that an expression is guaranteed to succeed the corresponding runtime check is a *static* property which can be controlled by the type system. More precisely, an expression of type $t\&P$ can either reduce to a value of the same type, which is guaranteed to succeed the check, or lead to a dynamic error, or not terminate, as it happens for any other type.

Note also that by the nominal interpretation of predicates we gain the well-known advantages of the nominal approach for software evolution: the meaning of a predicate can change, and a system with structural approach would be fragile.

Our design is in principle language independent. To integrate and implement check expressions on top of an existing language, they could be translated, by a pre-processing step, into the application of the corresponding predicate, a function written in the source language. In an imperative language, analogously to what is required in other proposals, predicates should be *pure* functions, and an object could be ensured to invariantly satisfy a property only under some conditions, the simplest being that the object is deeply immutable, that is, all its fields are (recursively) immutable. More permissive conditions could be allowed by combining the type system with modifiers for aliasing and immutability control, see, e.g., [6,5]. Interaction with polymorphic types is an interesting topic for future work. Instantiating polymorphic types with tagged types seems to pose no problems. Moreover, at a first sight, polymorphic types with tags seem to make sense in combination with subtyping constraints, e.g., a predicate `isempty` which holds when the argument, assumed to be of a subtype of `Collection`, is empty.

As future work, we also discuss two extensions.

Predicate subtyping Certain predicates may be stronger than others, for example we expect `greaterThan5` to imply `positive`. Since these predicates are defined as code, there is no general way to discover such implications automatically. However, we could offer a language construct for subtyping relations declared by the programmer, as usual for nominal types, for example, using again a Java-like syntax:

```
bool positive(int x) {return x>=0;}
bool greaterThan5(int x) {return x>5;}
  implies positive
```

This extension could be encoded in the original language by expanding occurrences of the stronger predicate name to a sequence, for instance:

```
int&greaterThan5 x=(greaterThan5)34
```

would be expanded to

```
int&greaterThan5&positive x=(greaterThan5)(positive)34
```

Refinement for pre-existing operations One limitation of the proposed approach is that existing operations are unaware of newly introduced properties.

For example, assume to have a `Sum` operation encoding the sum of two numbers, with type `int*int->int`. If we add the concept of `positive`, we know that, in a context where `a` and `b` are positive, `Sum(a,b)` should be positive; but in the current system it is just an `int`. An extension of our approach could allow:

```
SumPosPos refines Sum:
  int&positive*int&positive->int&positive
```

This extension could be encoded in the original language by declaring an auxiliary function `SumPosPos` which just calls `Sum` and casts the result to `int&positive`. In the case of an overpermissive refinement, the error can be traced back to the declaration of `SumPosPos`.

Acknowledgments. We are grateful to the anonymous reviewers for their useful suggestions, which led to substantial improvements.

References

1. A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In T. Ball and M. Sagiv, editors, *ACM Symp. on Principles of Programming Languages 2011*, pages 201–214. ACM Press, 2011.
2. K. Arkoudas and M. C. Rinard. Deductive runtime certification. *Electronic Notes in Theoretical Computer Science*, 113:45–63, 2005. Fourth Workshop on Runtime Verification (RV 2004).
3. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS'04 - Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
4. T. Freeman and F. Pfenning. Refinement types for ML. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 268–277. ACM, 1991.
5. P. Giannini, M. Servetto, and E. Zucca. Types for immutability and aliasing control (extended abstract). In *ICTCS'16 - Italian Conf. on Theoretical Computer Science*, 2016. In this volume.
6. C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2012)*, pages 21–40. ACM Press, 2012.

7. J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In M. T. Morazán, editor, *TFP 2007 - Trends in Functional Programming*, volume 8 of *Trends in Functional Programming*, pages 54–70. Intellect, 2007.
8. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, 2005.
9. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *VMCAI'16 - Verification, Model Checking, and Abstract Interpretation*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.
10. N. Nystrom, V. A. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In G. E. Harris, editor, *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2008)*, pages 457–474. ACM Press, 2008.
11. D. J. Pearce and L. Groves. Designing a verifying compiler: Lessons learned from developing Whiley. *Science of Computer Programming*, 113:191 – 220, 2015.
12. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In G. Castagna, editor, *ESOP 2009 - European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009.