Translation Correctness for First-Order Object-Oriented Pattern Matching

Burak Emir[1]*, Qin Ma[2], and Martin Odersky[1]

[1] EPFL, 1015 Lausanne, Switzerland
[2] OFFIS, Escherweg 2, Oldenburg, Germany

LAMP-REPORT-2007-003

Abstract

Pattern matching makes ML programs more concise and readable, and these qualities are also sought in object-oriented settings. However, objects and classes come with open class hierarchies, extensibility requirements and the need for data abstraction, which all conflict with matching on concrete data types. Extractor-based pattern matching has been proposed to address this conflict. Extractors are user-defined methods that perform the task of value discrimination and deconstruction during pattern matching. In this paper, we give the first formalization of extractor-based matching, using a first-order object-oriented calculus. We give a direct operational semantics and prove it sound. We then present an optimizing translation to a target language without matching, and prove a correctness result stating that an expression is equivalent to its translation.

1 Introduction

Algebraic datatypes and pattern matching render ML programs more concise, easier to read, and amenable to mathematical proof by structural induction [1]. Match expressions are high-level constructs with good properties: Compilers can translate them very efficiently [2–6]. However, ML style pattern matching is often incompatible with data abstraction.

To see why, consider the ML **datatype** definition in Fig. 1 which introduces a type name and its constructors. Constructors play the double role of tags and functions that aggregate data. Every list instance is tagged with either Nil or Cons. In match expressions, values are distinguished by their tag to recover their data. In the example, the append function concatenates lists by *matching* its argument xs: if it is the empty list, we return the second argument zs. If xs is a Cons cell, a non-destructive update appends zs to its tail via a recursive call.

This style of programming is concise and readable. The programs can be efficient, since tags can be translated to integers and match expressions to cascaded switch statements. However, note that the set of constructors as well as the arrangement of data items is fixed once and for all. Furthermore, the grouping of data items for the Cons constructor (e.g. **int** $*$ list) exposes the representation of the data, making it harder to change.

* present address: Google Switzerland GmbH, Freigutstr. 12, 8002 Zürich, Switzerland

```
datatype list = Nil | Cons of int * list

fun append (xs,zs) = case xs of
  Nil         ⇒ zs
| Cons(y,ys)  ⇒ Cons(y,append(ys,zs));
```

Fig. 1. Algebraic Matching in ML

Representation independence forms the basis for data abstraction. Researchers have suggested to reconcile pattern matching and data abstraction [7, 8]. However, slow adoption suggests that data abstraction, while desirable, is not considered essential for functional programming. In object-oriented programming, data abstraction is essential: it is one of the principles that permit object-oriented programmers to describe systems using class hierarchies.

Pattern matching in object-oriented systems is appropriate when the set of operations cannot be anticipated (cf. the Visitor design pattern [9]). Like functional pattern matching, the Visitor pattern breaks encapsulation: the internal object representation has to be exposed in order for operations like append to access them.

In order to use pattern matching without tying it to a closed set of types that expose their representation, a semantics for pattern matching based on user-defined functions has been proposed independently by Emir, Odersky and Williams [10], and Syme, Neverov and Margetson [11]. A pattern is interpreted as the invocation of a so-called extractor method, which discriminates and deconstructs the input value. This decouples the type of an accepted value from the representation extracted from it.

A short example of matching with extractors is given in Fig. 2. It defines a List class, three subclasses Nil, Cons, and Singleton as well as two extractors cons and nil. These extractors are methods, which here use test expressions $a?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\}$. Such a test expression is the contraction of the Java code

$$\textbf{if}\ (a\ \ \textbf{instanceof}\ \ C)\ \{\ C\ \mathsf{x} = (C)a; d\ \}\ \textbf{else}\ e\ .$$

The append implementation demonstrates the use of extractors. A pattern invokes an extractor method with an implicit argument — the $\mathsf{List}().\mathsf{cons}(\mathsf{y}, \mathsf{ys})$ pattern expresses two steps:

1. invoke method List().cons with the input value xs of the match expression as *single* argument
2. if the method returns **null**, the input value is rejected. Otherwise, it is accepted: bind the first field of the result to y and the second one to ys.

The return value of the extractor is a representation object that groups data items, which can be matched against subpatterns or bound to local variables. The type Cons is the result type of method cons, so it is used internally and externally to represent lists. Representation can be chosen independently though: note that lists that contain only one element can internally be represented with class Singleton, which is never exposed through pattern matching. This class could have been added later, without breaking client code that references cons and nil extractors.

We are interested in the question whether algorithms for optimized translation of pattern matching can be applied to extractors and the object-oriented context. Also, the question arises

```
class List() ◁ Obj {
  def nil(x: Obj): Nil = {                                  /* extractor */
    x?{y: Nil ⇒ y}/{null}
  }
  def cons(x: Obj): Cons = {                                /* extractor */
    x?{y: Cons ⇒ y}/{x?{y: Singleton ⇒ Cons(y.i, Nil())}/{null}}
  }
}
class Nil() ◁ List {}
class Cons(hd: int, tl : List) ◁ List {}
class Singleton(i: int) ◁ List {}        /* internal representation class */

def append(xs: List, zs: List): List = {
  xs match {
    case List().nil() ⇒ zs
    case List().cons(y, ys) ⇒ Cons(y, append(ys, zs))
}}
```

Fig. 2. Extractor-based Matching in the FPat Calculus

which conditions (if any) have to be satisfied by extractors in order to prove that optimizing translation preserves the meaning of the program.

In this paper, we present possible answers to these questions. For this purpose, we adapted the translation to decision-trees described by Pettersson [4] to extractors. Using a formal calculus based on Featherweight Java (FJ) [12], we define a first-order object-oriented calculus FPat that offers runtime type inspection and pattern matching. A generic version is presented elsewhere [13]. We give a direct operational semantics and then show a straightforward translation algorithm to compile match expressions down to the fragment without pattern matching. This translation is proven correct, under the hypothesis that extractor methods do not diverge and do not throw exceptions. To the best of our knowledge, this hypothesis was never mentioned in the literature, though its necessity is easily justified in our development.

Extractor-based pattern matching has been implemented independently in F♯ (there called "active recognizer") and Scala. In contrast to previous work [10], this paper aims to shed light on its formal underpinnings. In summary, we contribute:

- a formal calculus that precisely describes extractor-based pattern matching,
- a formal definition and correctness proof of an optimized translation of match expressions,
- and formal conditions extractors have to satisfy in order for optimization to be correct.

The rest of the paper is organized as follows. We define FPat in Section 2. We present the translation of pattern matching in Section 3. We describe the correctness proof in Section 4. Section 5 discusses related work and Section 6 concludes.

2 An Object-Oriented Calculus with Pattern Matching

The syntax and operational semantics of FPat are given in Fig. 3, and the typing rules and auxiliary definitions are given in Fig. 4 and Fig. 5. The calculus is based on FJ, but with

semantics defined using a strict, left-to-right big-step semantics. We will briefly review the definitions. We then define divergent programs and show type soundness.

2.1 Syntax

What follows is a short presentation of the rules and notation. A sequence $\alpha_1..\alpha_n$ is abbreviated as ${\alpha_\star}^{1..n}$, where α can be an expression, a name-type binding, or a judgment. The empty sequence is written $\bullet$. Multiple occurrence of the $\star$ indicate that the same index appears at multiple positions. Moreover, we shall need to express sequences with holes, so $\alpha_\star, \beta, {\alpha_\star}^{1..]i[..n}$ stands for $\alpha_\star^{1..i-1}, \beta, \alpha_\star^{i+1..n}$.

An FPat program ${cd_\star}^{1..n}; e$ is a set of class definitions and a top-level expression. Class definitions are kept in a class table, which we leave implicit throughout the paper and which satisifies the important properties that inheritance cycles and duplicate entries are absent. Classes have an explicit superclass as well as field declarations and method definitions, all publicly accessible. Methods can have an $@\mathbf{safe}$ annotation to indicate that they terminate without throwing an exception. The class hierarchy induces a subtype relation $<:$, of which the magic class Obj forms the largest element and the magic class Exc forms the smallest. These two types are magic because they do not have definitions in the class table. We also have a least upper bound $C \sqcup D$ operation, which is the least type E in the hierarchy that satisfies $C<:E$ and $D<:E$.

There are 8 expression forms: $\mathbf{null}$, variables x, field selection $e.\mathsf{f}$, method invocation $e.\mathsf{m}({e_\star}^{1..n})$, object construction $C({e_\star}^{1..n})$, exception $\mathbf{throw}$, test expressions $a?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\}$ and match expressions ${e_\star}^{1..n}\ \mathbf{match}\ \{{c_\star}^{1..k}\}$. The calculus does not model assignment nor object identity. The free variables fv and the defined variables dv are defined in the straightforward manner.

2.2 Semantics

We briefly describe operational semantics of the fragment without pattern matching, in order to be self-contained. Semantics specific to pattern matching are deferred to a separate section below.

Terminating computation of meaningful expressions is modeled by a big-step evaluation relation $e \Downarrow q$ that takes expressions e to results q. A result q is either a value $v \in \mathsf{Values}$, the $\mathbf{null}$ result or the exception $\mathbf{throw}$. Note that substitutions are all restricted to map variables only to values or $\mathbf{null}$. A dotted metavariable $\dot{v}$ indicates either a value v or $\mathbf{null}$.

A value is the outcome of an object construction $C({\dot{v}_\star}^{1..n})$, which is written without $\mathbf{new}$. There is no explicitly declared constructor, instead the field order determined by the inheritance hierarchy (specified in the auxiliary judgment $\mathsf{fields}(C)$) is used. The following correct program illustrates how arguments in object construction relate to fields in class definitions:

$$\begin{array}{l}\mathbf{class}\ D(\mathsf{f}\colon A)\ \triangleleft\ \mathsf{Obj}\ \{\ldots\}\\ \mathbf{class}\ C(\mathsf{g}\colon B)\ \triangleleft\ D\ \{\ldots\}\\ C(A(), B())\end{array}$$

Rules (Rfld), (Rinvk), (Rnew) in Fig. 3 describe field access, method invocation and object construction. The auxiliary judgment $\mathsf{mbody}(\mathsf{m}, C)$ specifies how to lookup method bodies. Rules (Cfld), (Crcv), (Carg), (Cnew) throw or propagate exceptions.

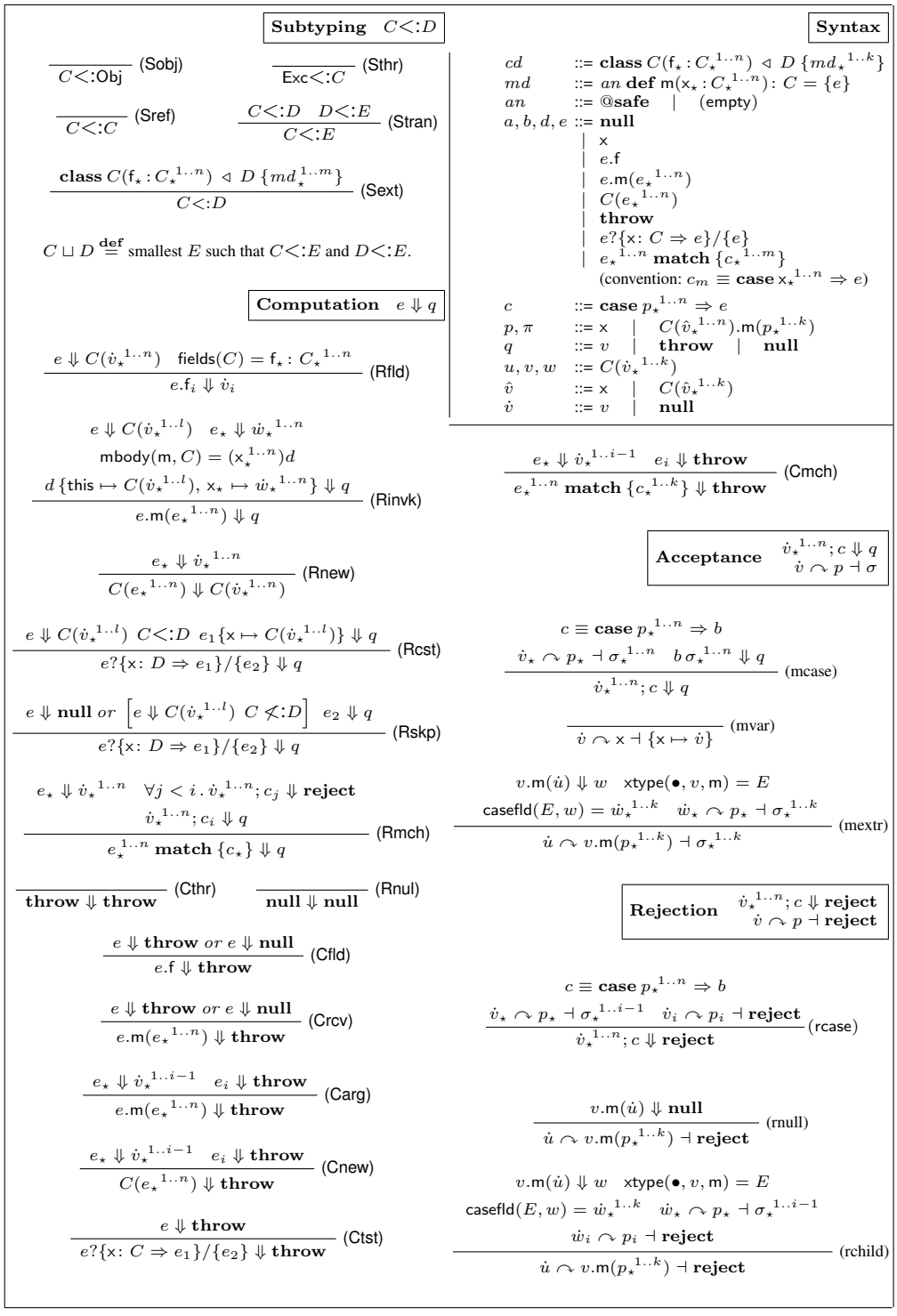

Fig. 3. FPat Syntax and Semantics

The only significant use of **null** happens in test expressions. Their behavior is specified in rules (Rcst) and (Rskp): if the tested expression, or *scrutinee*, is not **null** and its type is lesser than the required type, it is bound to a local variable and the first branch is evaluated (Rcst). Otherwise, the second branch is evaluated (Rskp). If the scrutinee throws, the exception is propagated (Ctst).

The relation $\Downarrow$ does not specify the behavior of meaningless or non-terminating programs. To show type soundness, divergent programs are defined using a relation $\Uparrow$ in Fig. 6. Meaningless expressions are then precisely those that neither terminate nor diverge.

2.3 Semantics of Matching

Pattern matching expressions contain one or more case clauses, each of which compares the n input values against n patterns. The last clause may only have variable patterns. This excludes pathological expressions of the form $e_\star{}^{1..k}$ $\mathbf{match}$ $\{\}$ and ensures that the behavior is defined for any possible combination of input values. It is easy to enforce this convention in a compiler.

Matching depends on judgments describing acceptance and rejection of patterns and cases. Rule (Rmch) describes evaluation of cases according to the first match policy: an accepting case is evaluated only if all preceding cases rejected the input.

Two separate judgments describe acceptance for cases $\dot{v}_\star{}^{1..n}; c \Downarrow q$ and patterns $\dot{v} \curvearrowright p \dashv \sigma$. We explain the judgments for case clauses first. A case accepts and evaluates to result q if each input value is accepted by the corresponding pattern (mcase). Analogously, a case rejects $\dot{v}_\star{}^{1..n}; c \Downarrow \mathbf{reject}$ if an initial segment of patterns accept and the following pattern rejects its input (rcase). Together, these rules describe a left-to-right evaluation of patterns. If a pattern accepts, it yields a substitution, and if all patterns accept, the combined substitution is applied to the body of the case (the merging of substitutions is indicated by juxtaposition).

The judgment $\dot{v} \curvearrowright p \dashv \sigma$ describes that pattern p accepts $\dot{v}$ and yields substitution σ. Analogously, the judgment $\dot{v} \curvearrowright p \dashv \mathbf{reject}$ describes rejection. A variable pattern always accepts its input (mvar), yielding the obvious substitution. An extractor pattern $C(\hat{v}_\star{}^{1..n}).\mathsf{m}(p_\star{}^{1..n})$ accepts (mextr) if:

1. evaluation of the extractor call returns a value w, yielding so-called case fields $\dot{w}_\star{}^{1..k}$
2. all subpatterns accept the case fields, yielding substitutions $\sigma_\star{}^{1..k}$

The extractor pattern rejects if the call returns **null** (rnull) or if one of its subpattern rejects its input (rchild). Case fields $\mathsf{casefld}(E, w)$ are determined for the return type E of the extractor method, as specified in the auxiliary judgment $\mathsf{xtype}(\bullet, \hat{v}, \mathsf{m})$. They are the fields declared in the class definition of E itself. Note that we will often abbreviate $C(\hat{v}_\star{}^{1..n})$ with $\hat{v}$.

The outcome is undefined when extractors throw exceptions or diverge. For this reason, the **@safe** annotation is required on any method that is referenced as an extractor. Safety in the above sense is an undecidable property or programs, but restrictions and approximations are available to tackle this problem. Our focus in this paper is on justifying the condition, not checking it. Avoiding exceptions is also the reason for excluding extractor calls $\mathsf{x.m}(\ldots)$.

Discussion In any statically-typed definition of pattern matching, order and types of subpatterns must be specified. The benefit of using extractors lies in decoupling the matched type from the representation type.

Pattern matching usually includes matching on literals like $42, \mathbf{true}$ and named constants like foo. While literal expressions are constructors without arguments, a corresponding convention for extractors can be assumed. Named constants are added by allowing tests for *singleton types* v.type. Structural equality then ensures that $C(\dot{w}_\star^{1..k}) \in v.\mathsf{type}$ if $v \equiv C(\dot{w}_\star^{1..k})$.

2.4 Typing

The FPat type system is specified through a set of syntax-directed typing rules in Fig.4. The rules specific to matching are described in a separate section below.

Type judgments for expressions have the form $\Gamma \vdash e \in C$ where Γ is a type environment (a finite mapping from variables to types), e an expression and C a class. The judgments $cd \diamond$ and $an\ md\ \diamond$ in C assert well-typedness of class and method definitions. Methods annotated with $@\mathbf{safe}$ are assumed to terminate and never throw exceptions for any input (including $\mathbf{null}$). A class definition is well-typed if all its methods are well-typed, and a method is well-typed if its return expression is well-typed under the appropriate type environment. If the method overrides a method in a superclass, their signatures have to be identical, which is asserted by the judgment $\mathsf{override}(an(B_\star^{1..n})B, \mathsf{m}, D)$. A program is well-typed if all its class definitions are well-typed, and its top-level expression is well-typed in the empty environment.

Typing expressions is straightforward. Rules (Tthr) and (Tnul) give the most specific type Exc to the $\mathbf{throw}$ and $\mathbf{null}$ results. (Tvar) takes the type of a variable from the type environment, and field access (Tfld) and object construction (Tnew) is checked against the fields of the class as calculated by the judgment $\mathsf{fields}(C)$. A similar judgment for method signatures $\mathsf{mtype}(\mathsf{m}, C)$ is used to type-check method invocation (Tinvk). Thus, well-typed method calls and objects constructions have the right number and types of arguments.

Test expressions are checked with rule (Ttst), which modifies the type environment for the succeeding branch to account for the new local variable. Binding in test expressions can be used to define a derived form $\mathbf{val}\ \mathsf{x}\colon C = a; b$, which we will introduce in Section 3.1.

Some readers may notice that we allow test expressions that are statically known to fail, such as $C(\ldots)?\{\mathsf{y}\colon D \Rightarrow \ldots\}/\{\ldots\}$ for unrelated C, D. The reason for not checking the static type against the type to be tested is that it would make it impossible to prove a substitution lemma (FJ has "stupid casts" for similar reasons): the expression to be tested might have a more precise type after substitution, and only then it would become apparent that the test expression fails. A real compiler would reject test expressions in source programs if it can derive at compile time that a test expression fails.

2.5 Typing of Match Expressions

Match expressions are well-typed if all their clauses are well-typed (Tmch), using the least upper bound to combine the clauses' result types. To type-check a single clause $\mathbf{case}\ p_\star^{1..n} \Rightarrow b$ (Tcase), each pattern in $p_\star^{1..n}$ is type-checked w.r.t. the type environment Γ and an "expected type" $C_\star$, yielding a type environment $\Delta_\star$ as in $\Gamma; C_\star \ni p_\star \dashv \Delta_\star^{1..n}$. Then, the body is type-checked against the combined type environments as in $\Gamma, \Delta_\star^{1..n} \vdash b \in D$. Variables introduced in patterns must be pair-wise different and may not clash with Γ, which is implicit in the juxtaposition of environments.

Pattern typing $\Gamma; E \ni p \dashv \Delta$ is type-checked as follows: For variable patterns (TPvar), the expected type E is used to produce a singleton environment. For extractor patterns (TPextr),

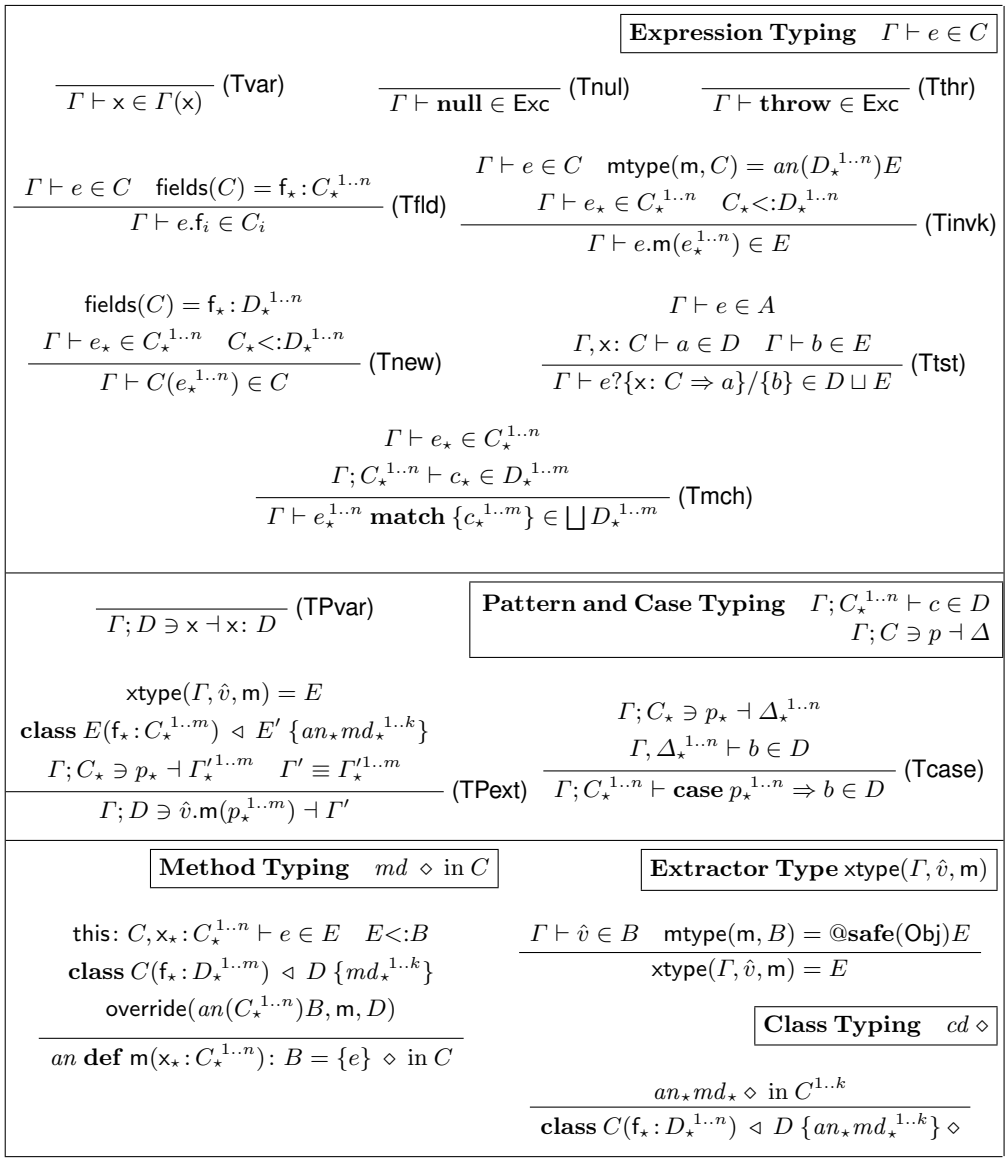

Fig. 4. Typing Rules

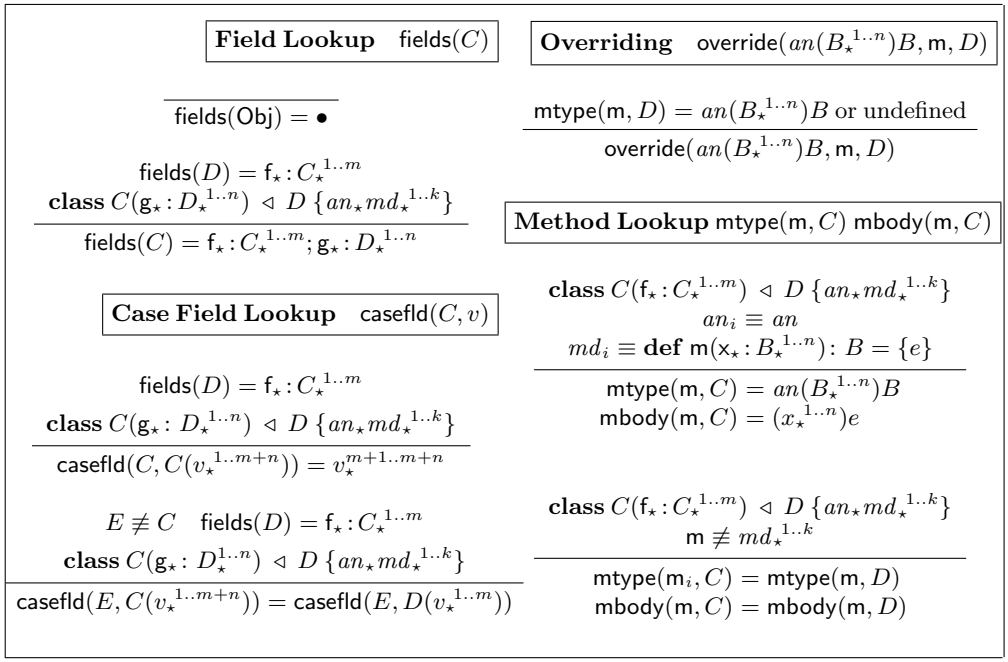

Fig. 5. Auxiliary Definitions

the judgment $\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m})$ looks up the type of receiver and the signature of the extractor method in order to recover the representation type. It also ensures that extractors are **@safe**. The case field types are then used as expected types to check the subpatterns. Finally, the environments $\Delta_\star^{1..m}$ obtained from the subpatterns are merged into one environment Δ.

2.6 Divergent Programs

Proving type soundness for big-step semantics necessitates specifying in some way the meaningless programs ruled out by the type system. One possible approach is to define a special value **wrong** and characterize meaningless programs as those that are evaluated to **wrong**. This approach requires is error-prone, because if by mistake, a **wrong** rule is omitted, a misleading statement of "type soundness" is proven that does not actually exclude all meaningless programs.

A safer approach, suggested byOur approach to type soundness, following Leroy and Grall [14], is to characterize meaningless programs as those that neither terminate nor diverge. A forgotten rule in the specification of divergent programs would then make the proof of the big-step version of the "Progress" lemma impossible. For our purpose, specifying divergent programs has the additional advantage of being relevant to correctness of pattern matching translation (ideally, we want to avoid translating divergent programs into terminating ones). Divergent programs are defined coinductively by the set of divergence rules in Fig. 6. These rules are tailored to establish that any well-typed term that does not terminate necessarily diverges. Their coinductive nature is indicated by horizontal double lines: coinductive derivations are

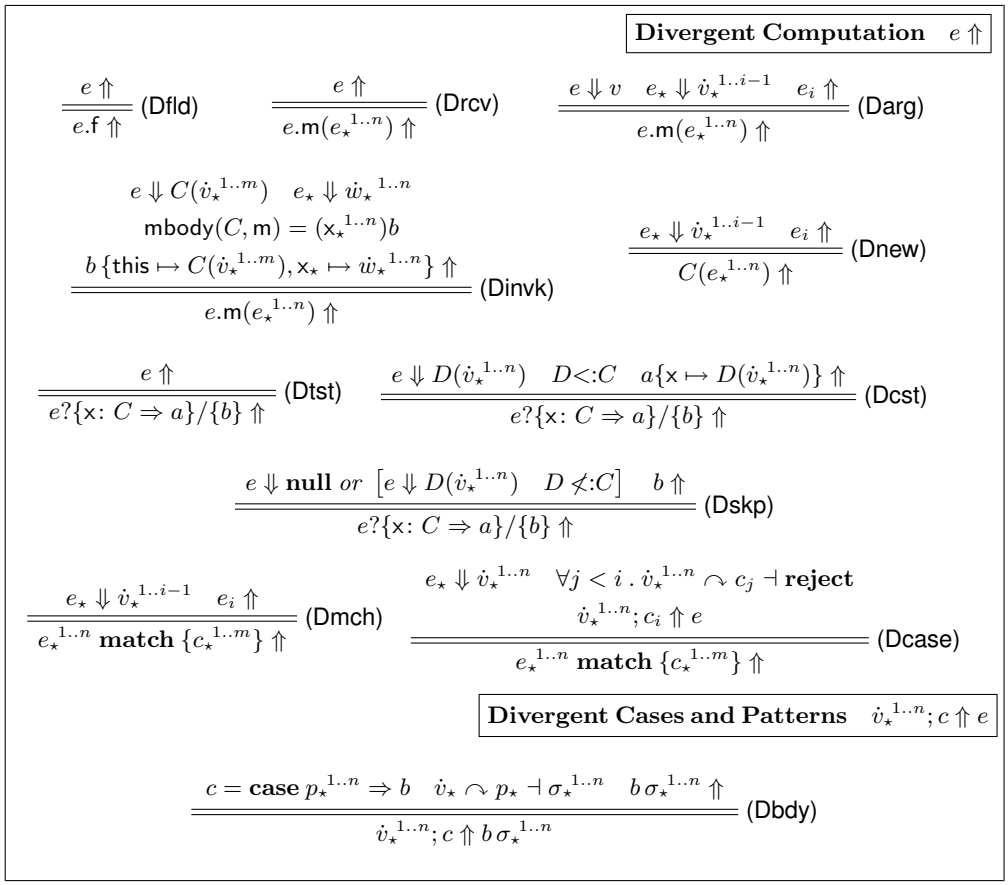

Fig. 6. Divergent Programs

infinite trees with the root being the assertion to derive and the successors of each node being determined by a derivation rule.

Let us consider the rules one by one. Rule (Dfld) and (Drcv) express that accessing a field or invoking a method on a divergent expression yields a divergent expression. Rules (Darg) and (Dnew) say that object construction and method invocation diverges if one of their arguments diverge. Note that a strict call-by-value, left-to-right evaluation order is followed also here. Rule (Dinvk) says that calling a method with arguments that make the method body diverge yields a divergent expression. Similarly, (Dtst), (Dcst) and (Dskp) characterize divergent test expressions by locating divergence in the respective subexpression.

In match expressions, a divergent match expression can be traced back to some (possibly empty) initial segment of rejecting case clauses followed by a divergent case clause. A case clause may only diverge because its body diverges (Dbdy).

Discussion If we did not rely on the @**safe** annotation, divergence could also be caused by extractor calls during pattern match evaluation. It is possible but tedious to omit the @**safe**

hypothesis, give divergence and exception propagation rules for extractor calls and adapt the soundness proof accordingly. Moreover, we discover that **@safe** is actually needed for correctness of the optimizing translation (see section 3.2). As a consequence, we chose the simpler route by defining only those divergent programs that are needed for **(Progress)**. Here and in the proofs, bold-face names and phrases in parentheses refer to lemmata and theorems.

We recall the principle of coinduction, that lets us prove for a set of expressions that all of them diverge. Consider the following class definition and the expression $\mathsf{Lp}().\mathsf{m}()$ diverges:

$$\mathbf{class}\ \mathsf{Lp}\ \{\ \mathbf{def}\ \mathsf{m}() : \mathsf{Lp} = \{\mathsf{this}.\mathsf{m}()\}\ \}$$

$$\frac{\mathsf{Lp}() \Downarrow \mathsf{Lp}() \quad \mathsf{this}().\mathsf{m}()\{\mathsf{this} \mapsto \mathsf{Lp}()\} \in R}{\mathsf{Lp}().\mathsf{m}() \in R}\ (\mathsf{Dinvk})$$

We pick the set $R = \{\mathsf{Lp}().\mathsf{m}()\}$. In order to show that $R \subseteq \Uparrow$ it is enough to show that it is preserved by the divergence rules. Since $\Uparrow$ is defined coinductively, it is by definition the largest possible set preserved by the divergence rules.

Preserving the set means that if a divergence rule is applied "backwards", i.e. from conclusion to the premises, it only takes elements in the set to other elements in the set. For $\mathsf{Lp}().\mathsf{m}()$, we observe that rule (Dinvk) considered backwards only yields the very same expression, because $\mathsf{this}().\mathsf{m}()\{\mathsf{this} \mapsto \mathsf{Lp}\} \equiv \mathsf{Lp}().\mathsf{m}()$. In other words, (Dinvk) preserves membership in R for $\mathsf{Lp}().\mathsf{m}()$. Since this is the only expression in R, the whole set is preserved by the divergence rules. Consequently, $R \subseteq \Uparrow$ and $\mathsf{Lp}().\mathsf{m}()$ is a divergent program.

2.7 Soundness

We now prove type soundness using big-step versions of the standard lemmata.

Lemma 1 (Uniqueness) *For all a, if $a \Downarrow q$ then for all q', if $a \Downarrow q'$ then $q = q'$.*

Lemma 2 (Termination) *For all a and all q, it holds that if $a \Downarrow q$ then $a \not\Uparrow$.*

Lemma 3 (Subtypes have all Fields) *If $C{<:}D$, $C \neq \mathsf{Exc}$ then $\mathsf{fields}(C) = \mathsf{fields}(D); \mathsf{g}_\star : E_\star^{1..m}$.*

Lemma 4 (Subtypes have all Methods) *If $C{<:}D$, $C \neq \mathsf{Exc}$ and $\mathsf{mtype}(\mathsf{m}, D) = an(C_\star^{1..n})B$, then $\mathsf{mtype}(\mathsf{m}, C) = an(C_\star^{1..n})B$.*

The following two lemmata are needed to prove the substitution lemma for pattern matching expressions. We have to deal with the typing rule for variables which might end up producing a "better" environment for input values whose type has become more precise after substitution. We write $\Delta'{<:}\Delta$ when $\mathsf{dom}(\Delta) = \mathsf{dom}(\Delta')$ and $\mathsf{x}\colon B \in \Delta$ implies $\mathsf{x}\colon A \in \Delta'$ with $A{<:}B$.

Lemma 5 (Subtypes yield Refined Environment)
If $C{<:}D$ and $\Gamma; D \ni p \dashv \Gamma'$ then $\Gamma; C \ni p \dashv \Gamma''$ for some $\Gamma''{<:}\Gamma'$.

Lemma 6 (Refined Environment preserves Typing)
If $C_\star{<:}D_\star^{1..n}$ and $\Gamma, \mathsf{x}_\star : D_\star^{1..n} \vdash e \in B$ then $\Gamma, \mathsf{x}_\star : C_\star^{1..n} \vdash e \in A$ for $A{<:}B$.

Lemma 7 (Weakening) *If $\Gamma \vdash d \in S$ and $\mathsf{x} \notin fv(d)$, then $\Gamma, \mathsf{x}\colon T \vdash d \in S$ for any T.*

Lemma 8 (Substitution Lemma) *If* $\Gamma, \mathsf{x}_\star : B_\star^{1..n} \vdash b \in D$ *and* $\bullet \vdash \dot{u}_\star \in A_\star^{1..n}$ *for* $A_\star <: B_\star^{1..n}$, $\dot{u}_\star \in \mathsf{Values} \cup \{\mathbf{null}\}$ *then* $\Gamma \vdash b\,\{\mathsf{x}_\star \mapsto \dot{u}_\star^{1..n}\} \in C$, *for* $C <: D$.

Lemma 9 (Preservation)
If $a \Downarrow q$ *and* $\bullet \vdash a \in C$, *then* $\bullet \vdash q \in C'$ *for some* $C' <: C$.

The big-step version of the progress lemma uses coinduction.

Lemma 10 (Progress)
If $\bullet \vdash a \in C$ *and* $a \not\Downarrow q$ *for all* q, *then* $a \Uparrow$.

Theorem 1 (Type Soundness)
If $\bullet \vdash a \in C$ *then either* $a \Uparrow$ *or* $a \Downarrow q$ *for some* q *with* $\bullet \vdash q \in C'$, $C' <: C$.

3 Translation

3.1 Rewriting Match Expressions

An elegant way to describe translation is to give a set of rewrite rules, which are applied successively until all match expressions are replaced with lower-level operations. Apart from being easy to understand and implement, correctness can then be established for each rule separately.

There are two approaches to the compilation of pattern matching, one based on decision-trees and the other based on backtracking automata [6, 5]. We chose the translation to decision trees, which in the functional setting guarantees that no input value is tested more than once. Our presentation of the algorithm follows Pettersson's [4].

The central idea is to remove a top-level pattern of a case clause, lifting its subpatterns to the top-level. Consider the two expressions below, for fresh $\mathsf{y}, \mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$

$$
\begin{array}{l}
//\mathit{recall}\ \mathsf{xtype}(\Gamma, \mathsf{List}(), \mathsf{cons}) = \mathsf{Cons} \\
\mathsf{x}\ \mathbf{match}\ \{ \\
\quad \mathbf{case}\ \mathsf{List}().\mathsf{cons}(\pi_1, \pi_2) \quad \Rightarrow a \\
\quad \mathbf{case}\ \mathsf{y}_0 \qquad\qquad\quad \Rightarrow b \\
\}
\end{array}
\qquad
\begin{array}{l}
\mathsf{List}().\mathsf{cons}(\mathsf{x})?\{\mathsf{y}\colon \mathsf{Cons} \Rightarrow \\
\quad (\mathsf{x}, \mathsf{y}.\mathsf{hd}, \mathsf{y}.\mathsf{tl})\ \mathbf{match}\ \{ \\
\qquad \mathbf{case}\ \mathsf{y}_1\ , \pi_1\ , \pi_2 \Rightarrow a \\
\qquad \mathbf{case}\ \mathsf{y}_0\ , \mathsf{y}_2\ , \mathsf{y}_3 \Rightarrow b \\
\}\}/\{\quad \mathsf{x}\ \mathbf{match}\ \{\mathbf{case}\ \mathsf{y}_0 \Rightarrow b\}\}
\end{array}
$$

Some scrutiny reveals that these are actually equivalent. The extractor of the first pattern $\mathsf{List}().\mathsf{cons}(\pi_1, \pi_2)$ in the first case has been pulled out and a test is done on the outcome: if it is non-null, it is bound to the fresh variable y and the subpatterns are matched against the case fields $\mathsf{y}.\mathsf{hd}, \mathsf{y}.\mathsf{tl}$. Note that the width of the original match is augmented by lifting the nested patterns to the top-level. Since π_1, π_2 can potentially reject the input, all cases of the original match are copied to the new one. Some entries need to be *expanded* to match the arity of the new match, which is done by using fresh variable patterns $\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$. If the extractor returns **null**, the first clause rejects and so the second branch of the test expression deals with the remaining cases of the match.

The algorithm performs optimization by reusing results of an extractor call: calls to the same extractor in the same column are replaced with clauses that match subpatterns (if the call succeeded), or discarded altogether (if the result was null). We illustrate the optimization with

Translation $[\![\Gamma \vdash a\ \mathbf{match}\ \{c_\star^{1..m}\} \in D]\!] = e$

(Tmp)

$$[\![\ \frac{\ldots \quad \Gamma \vdash a_\star \in C_\star^{1..n}}{\Gamma \vdash a_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \in D}\ ]\!] = \mathbf{val}\ \mathsf{z}_\star : C_\star = a_\star^{1..n}; \mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\}$$

condition: – $a_\star$ are not all variables

(Var)

$$[\![\ \frac{\ldots \quad \Gamma \vdash \mathsf{z}_\star \in C_\star^{1..n}}{\Gamma \vdash \mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \in D}\ ]\!] = b\,\{\mathsf{x}_\star \mapsto \mathsf{z}_\star^{1..n}\}$$

condition: – c_1 has the shape $\mathbf{case}\ \mathsf{x}_\star^{1..n} \Rightarrow b$

(Mix)

$$[\![\ \frac{\ldots \quad \Gamma \vdash \mathsf{z}_\star \in C_\star^{1..n}}{\Gamma \vdash \mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\} \in D}\ ]\!] = \hat{v}.\mathsf{m}(\mathsf{z}_i)?\{\mathsf{y} : C \Rightarrow \mathbf{val}\ \mathsf{y}_\star : D_\star = \mathsf{y}.\mathsf{f}_\star^{1..k}; d\}/\{e\}$$

condition: – c_1 has the shape $\mathbf{case}\ \mathsf{x}_\star^{1..i-1}\ \hat{v}.\mathsf{m}(p_\star'^{1..k})\ p_\star^{i+1..n} \Rightarrow b$

translation steps:

– $\hat{v}.\mathsf{m}(\mathsf{z}_i)$ has pattern typing $\dfrac{\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m}) = C \quad \ldots}{\Gamma; C_i \ni \hat{v}.\mathsf{m}(p_\star'^{1..k}) \dashv \Delta}$ (TPext)

– the definition of C is $\mathbf{class}\ C(\mathsf{f}_\star : D_\star^{1..k}) \lhd E\ \{md_\star^{1..n}\}$

– $\mathsf{y}, \mathsf{y}_\star^{1..k}$ are fresh variables

– $d = \mathsf{z}_\star, \mathsf{y}, \mathsf{y}_\star^{1..k}, \mathsf{z}_\star^{1..]i[..n}\ \mathbf{match}\ \{expand_{\hat{v}.m}(c_\star^{1..m})\}$

– $e = \mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{other_{\hat{v}.m}(c_\star^{1..m})\}$

– where $expand_{\hat{v}.m}$ and $other_{\hat{v}.m}$ are defined for arbitrary patterns $p_\star, p'_\star$, as (subscript omitted):

expand(•) = •

expand($\mathbf{case}\ p_\star\ \hat{v}.\mathsf{m}(p_\star^{1..k})\ p_\star^{1..]i[..n} \Rightarrow b;\ c_\star^{1..m}$) =
  $\mathbf{case}\ p_\star\ \mathsf{z}'\ p'_1\ \cdots p'_k\ p_\star^{1..]i[..n} \Rightarrow b$; *expand*($c_\star^{1..m}$) z'*fresh*

expand($\mathbf{case}\ p_\star\ p\ p_\star^{1..]i[..n} \Rightarrow b;\ c_\star^{1..m}$) =
  $\mathbf{case}\ p_\star\ p\ \mathsf{z}'_1\ \cdots \mathsf{z}'_k\ p_\star^{1..]i[..n} \Rightarrow b$; *expand*($c_\star^{1..m}$) $\mathsf{z}_\star'^{1..k}$*fresh* $p \neq \hat{v}.\mathsf{m}(p_\star^{1..k})$

other(•) = •

other($\mathbf{case}\ p_\star\ \hat{v}.\mathsf{m}(p_\star'^{1..k})\ p_\star^{1..]i[..n} \Rightarrow b;\ c_\star^{1..m}$) = *other*($c_\star^{1..m}$)

other($\mathbf{case}\ p_\star\ p\ p_\star^{1..]i[..n} \Rightarrow b;\ c_\star^{1..m}$) =
  $\mathbf{case}\ p_\star\ p\ p_\star^{1..]i[..n} \Rightarrow b$; *other*($c_\star^{1..m}$) $p \neq \hat{v}.\mathsf{m}(p_\star^{1..k})$

Fig. 7. FPat Translation Rules

an example.

$$
\begin{array}{l}
\mathsf{x}\ \mathbf{match}\ \{ \\
\quad \mathbf{case}\ \mathsf{List}().\mathsf{cons}(\pi_1, \pi_2) \Rightarrow a \\
\quad \mathbf{case}\ \mathsf{List}().\mathsf{nil}() \Rightarrow b \\
\quad \mathbf{case}\ \mathsf{List}().\mathsf{cons}(\pi_3, \pi_4) \Rightarrow d \\
\quad \mathbf{case}\ \mathsf{y}_0 \Rightarrow e \\
\}
\end{array}
\qquad
\begin{array}{l}
\mathsf{List}().\mathsf{cons}(\mathsf{x})?\{\mathsf{y} : \mathsf{Cons} \Rightarrow \\
\quad (\mathsf{x}, \mathsf{y.hd}, \mathsf{y.tl})\ \mathbf{match}\ \{ \\
\quad\quad \mathbf{case}\ \mathsf{y}_1 \quad , \pi_1\ , \pi_2 \Rightarrow a \\
\quad\quad \mathbf{case}\ \mathsf{List}().\mathsf{nil}()\ , \mathsf{y}_2\ , \mathsf{y}_3 \Rightarrow b \\
\quad\quad \mathbf{case}\ \mathsf{y}_4 \quad , \pi_3\ , \pi_4 \Rightarrow d \\
\quad\quad \mathbf{case}\ \mathsf{y}_0 \quad , \mathsf{y}_5\ , \mathsf{y}_6 \Rightarrow e \\
\}\}/\{ \\
\quad \mathsf{x}\ \mathbf{match}\ \{\ \mathbf{case}\ \mathsf{z}.\mathsf{nil}() \Rightarrow b \\
\quad\quad\quad\quad \mathbf{case}\ \mathsf{y}_0 \quad \Rightarrow e\ \}\}
\end{array}
$$

Here, the first and third case (on the left) test the same extractor $\mathsf{List}().\mathsf{cons}$. This extractor call has been pulled out into a test expression. If it succeeds (then-branch), the resulting value is deconstructed and matched against the subpatterns. Again, since patterns π_1, π_2 may fail, we include all other cases, but we do not need to repeat the extractor call. If the extractor call returns **null**, then (else-branch) the remaining test cases are those that have extractor patterns *other* than $\mathsf{List}.\mathsf{cons}$. This suggests an recursive algorithm that identifies common patterns and translates them into test expressions and new, smaller match expressions.

Figure 7 contains the rewrite rules used by the algorithm. The translation relies on the static types of expressions, and is thus expressed as a translation of type derivations. The rules use a derived form $\mathbf{val}\ \mathsf{x}\colon C = a; b$ which has the double purpose of simplifying the presentation and catching divergent and exception-throwing input values. The derived form is only used when a is of static type C, and abbreviates $a?\{\mathsf{x}\colon C \Rightarrow b\}/\{b'\}$ where $b' = b\,\{\mathsf{x} \mapsto \mathbf{null}\}$.

Rule (Tmp) introduces **val** definitions, so that input values are always variables. Rule (Var) handles matches that are known to accept. The essential rule is (Mix) which performs the optimizing translation described above. If the extractor returns value w, then the subvalues $\mathsf{casefld}(C, w)$ can be obtained with field accesses $w.\mathsf{f}_\star^{1..k}$, and the return value as well as the subvalues are bound to fresh local variables $\mathsf{y}, \mathsf{y}_\star^{1..k}$. The function $expand$ adapts the width of case clauses as mentioned before. If the extractor returns **null**, we continue matching on those clauses that have a different extractor, computed by function $other$.

In contrast to functional pattern matching, we cannot assume that e.g. a rejecting extractor cons means that nil will necessarily accept the input value. The user-defined methods could be annotated to supply this information, an extension that we do not pursue in this paper. We shall call $rewrite$ the function that applies a rule to a suitable term (with its typing derivation).

3.2 Why must extractors be @safe to allow optimized translation?

Recall the example above. Suppose π_1 was a variable pattern and π_2, π_4 test the same extractor. Optimizing for the failing pattern π_2 causes omission of the entire third case clause.

When omitting this case clause, we are already assuming that π_3 will either accept or reject its input. However, if π_3 were allowed to throw an exception or diverge, it would not be possible to omit its evaluation without changing the meaning of the program. For this reason, the semantics does not cover these anomalous situations (if we included them, we could not prove our optimization correct). Any semantics for pattern matching that involves user-defined code depends on this assumption if optimized translation of matching is to preserve the meaning of programs, since we usually do not expect divergent or exception throwing programs to turn into normally terminating ones. A correct translation without the **@safe** assumption

would have to include case clauses that are *known* to fail for the sole purpose of preserving their exception-throwing or divergent behavior.

The assumption that extractors are **@safe** complements the assumptions formulated by Syme *et al* [11] and Okasaki [8] that informally require extractors to be side-effect free and return the same result in all execution contexts in order for optimization to work. Of course in this calculus, absence of side-effects is guaranteed by the absence of assignment.

3.3 The Algorithm

We define a function *transform* that recursively traverses expressions, rewriting any match statements it finds.

$$
\begin{array}{ll}
\mathit{transform}(\mathbf{null}) & = \mathbf{null} \\
\mathit{transform}(\mathsf{x}) & = \mathsf{x} \\
\mathit{transform}(e.\mathsf{f}) & = \mathit{transform}(e).\mathsf{f} \\
\mathit{transform}(e.\mathsf{m}({e_\star}^{1..n})) & = \mathit{transform}(e).\mathsf{m}({e'_\star}^{1..n}) \\
 & \qquad \text{where } e'_\star = \mathit{transform}(e_\star)^{1..n} \\
\mathit{transform}(\mathbf{throw}) & = \mathbf{throw} \\
\mathit{transform}(a?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\}) & = a'?\{\mathsf{x}\colon C \Rightarrow d'\}/\{e'\} \\
 & \qquad \text{where } a' = \mathit{transform}(a) \\
 & \qquad \text{and } d' = \mathit{transform}(d) \\
 & \qquad \text{and } e' = \mathit{transform}(e)
\end{array}
$$

$$
\mathit{transform}({e_\star}^{1..n}\ \mathbf{match}\ \{{c_\star}^{1..k}\}) = \mathit{transform}(\mathit{rewrite}({e_\star}^{1..n}\ \mathbf{match}\ \{{c_\star}^{1..k}\}))
$$

The *transform* function is then naturally extended to method definitions and class definitions. A program is translated by translating all class definitions and the top-level expression. Note that a single application of a rewrite rule takes place in one of the following contexts:

Definition 1 (Target Context) *A target context is defined by the following grammar:*

$$
\begin{array}{l}
\xi, \zeta ::= [\,] \mid \xi.\mathsf{f} \mid \xi.\mathsf{m}({b_\star}^{1..n}) \mid a.\mathsf{m}(b_\star, \xi, {b_\star}^{1..]i[..n}) \\
\qquad \mid \xi?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\} \mid a?\{\mathsf{x}\colon C \Rightarrow \xi\}/\{e\} \mid a?\{\mathsf{x}\colon C \Rightarrow d\}/\{\xi\}
\end{array}
$$

By the reasoning in the next section, this rewrite preserves the meaning of the program. A subsequent call of *transform* performs the same for subexpressions of a', until all match expressions are translated away.

4 Correctness

We define a formal notion of equivalence. Recall that a substitution always satisfies $\mathsf{x}\sigma \equiv \mathbf{null}$ or $\mathsf{x}\sigma \in \mathsf{Values}$ for all $\mathsf{x} \in \mathsf{dom}(\sigma)$. We proceed in two steps, following the *démarche* of [15]: we define a notion of equivalence and show that it is stable under contexts. Then we show that an expression is equivalent to its translation.

4.1 Equivalence and Open Equivalence

Definition 2 (Equivalence) *For d, e expressions with $fv(d) = fv(e) = \emptyset$, d is equivalent to e (written $d \approx e$), if both of these conditions hold: 1. for all q, if $d \Downarrow q$ then $e \Downarrow q$, and 2. if $d \Uparrow$ then $e \Uparrow$.*

Showing that $\approx$ is an equivalence relation is easy using **(Uniqueness)**,**(Termination)**. Equivalence alone is not enough for our purpose, since rewrite rules take place in context. We now define an equivalence on open terms and show it is stable under contexts.

Definition 3 (Open Equivalence) *For expressions d, e with $fv(d) \cup fv(e) \subseteq X$, d is open-equivalent to e (written $X \Vdash d \approx e$) if $d\sigma \approx e\sigma$ for all substitutions σ with $X \subseteq \mathsf{dom}(\sigma)$.*

Lemma 11 (Substitution preserves Equivalence) *If $X \Vdash d \approx e$, then for any substitution σ with $\mathsf{dom}(\sigma) \subseteq X$, it holds that $X \backslash \mathsf{dom}(\sigma) \Vdash d\sigma \approx e\sigma$.*

Theorem 2 (Congruence) *If $X \Vdash d \approx e$, then $Y \Vdash \xi[d] \approx \xi[e]$ for $Y = fv(\xi[d]) \cup fv(\xi[e])$.*

We now have everything we need for proving the correctness theorem. Since equivalence is a congruence, it is enough to show correctness of the rewrite rules. The proof of the following theorem references the **@safe** assumption, to derive case clause rejection for clauses omitted by $other_{\hat{v}.\mathsf{m}}$ – this requires normal termination of extractor calls to the left of the $\hat{v}.\mathsf{m}$.

Theorem 3 (Correctness of $[\![\,]\!]$) *For $a \equiv a_\star{}^{1..n}$ $\mathbf{match}$ $\{c_\star{}^{1..m}\}$, $fv(a) = X$, typing $\Gamma \vdash a \in A$, translation $a' = [\![\Gamma \vdash a \in A]\!]$ it holds that $X \Vdash a \approx a'$.*

Corollary 1 (Complete Algorithm) *The algorithm described in Section 3.3 is correct.*

Proof Consequence of the above theorem, applied sequentially to every application of a rewrite rule $[\![\,]\!]$, and transitivity of $\approx$. For termination, observe that each match expression produced by a rewriting rule is smaller than the original match expression using the lexicographically ordered tuples $\langle i, j, k\rangle$ where i is the number of non-variable input values, j the number of case clauses, and k the number of extractor patterns in $c_\star{}^{1..j}$. This ordering shows that for any e, all chains of dependency-pairs $\langle transform(e), transform(rewrite(e))\rangle$ must be finite. □

5 Related Work

Pettersson [4] and Ramsay and Scott [5] describe a matrix-based algorithm for translating match expressions to decision trees (the latter allowing heuristics other than left-to-right). Since an algebraic data type defines a closed set of constructors, different optimizations are available. Maranget [16] treats clause matrices and incompleteness checking in more detail.

If extractors came with coverage annotations, then more optimizations and incompleteness checking could be integrated. Syme, Neverov and Margetson [11] use *structured names* for this purpose. The authors also introduce parameterized patterns and give strong informal guidelines to restrict extractors (there called recognizers) in order to allow optimizations.

Extractors are rooted in Wadler's work on views [7], which Okasaki adapted to ML [8]. The design in a functional language context that comes closest to ours is Gostanza [17]. A more detailed discussion of the literature on views is presented in [10, 13].

Zenger and Odersky use pattern matching and algebraic datatypes in an object-oriented setting to handle the extensibility problem [18]. Liu and Myers add pattern matching to a Java like language by introducing forwards and backwards modes of evaluation [19].

6 Conclusion

We presented a formal object-oriented calculus with pattern matching. We proved the calculus sound, and gave an optimizing translation algorithm. We then proved the translation correct, revealing an important assumption required for correctness: that extractor patterns may not throw exceptions or diverge. We emphasize that non-optimizing translation is not affected by this requirement – yet, optimizing pattern matching by factoring out common test seems essential for good performance.

In future work, we would like to extend our formalization to support incompleteness checking and further optimizations as known from algebraic data types. The Scala language offers matching on specific types (case classes) and the **sealed** keyword to this end. Specifying the completeness of a set of extractors for a given domain would be possible through source annotations. Apart from this, further study is necessary to analyze how the backtracking approaches to pattern match translation can be adapted.

References

1. Burstall, R.M.: Proving properties of programs by inductive definitions. Computer (1969) 41–48
2. Wadler, P.: Pattern Matching, Chapter 4 of Peyton Jones, Wadler "Implementation of Functional Programming Languages", Prentice Hall. (1987)
3. Field, A., Harrison, P.: Functional Programming. Addison Wesley (1988)
4. Pettersson, M.: A term pattern-match compiler inspired by finite-automata theory. In: Proc. of International Workshop on Compiler Construction (CC). Volume Volume 641 of LNCS. (1992)
5. Scott, K., Ramsey, N.: When do match-compilation heuristics matter? Technical Report CS-2000-13, University of Virginia (2000)
6. Fessant, F.L., Maranget, L.: Optimizing pattern matching. In: Proc. of International Conference on Functional Programming(ICFP). (2001) 26–37
7. Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: Proc. of Principles of Programming Languages (POPL). (1987)
8. Okasaki, C.: Views for Standard ML. In: Proceedings of SIGPLAN Workshop on ML. (1998) 14–23
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1995)
10. Emir, B., Williams, J., Odersky, M.: Matching objects with patterns. In: Proc. of European Conference on Object-Oriented Programming (ECOOP). (2007)
11. Syme, D., Neverov, G., Margetson, J.: Extensible Pattern Matching via a Lightweight Language Extension. In: Proceedings of International Conference on Functional Programming (ICFP). (2007)
12. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA). (1999)
13. Emir, B.: Object-Oriented Pattern Matching. PhD thesis, EPFL Lausanne (2007)
14. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Theoretical Computer Science ((submitted))
15. Ma, Q.: Concurrent Classes and Pattern Matching in the Join Calculus. PhD thesis, INRIA-Rocquencourt and University Paris 7 (2005)
16. Maranget, L.: Warnings in pattern matching. Journal of Functional Programming **17**(3) (2007) 387–421
17. Gostanza, P.P., Pena, R., Nunez, M.M.: A new look at pattern matching in abstract data types. In: Proceedings of International Conference on Functional Programming (ICFP). (1996)
18. Zenger, M., Odersky, M.: Extensible algebraic datatypes with defaults. In: Proceedings of the International Conference on Functional Programming (ICFP). (2001)
19. Liu, J., Myers, A.C.: JMatch: Iterable Abstract Pattern Matching for Java. In: Proceedings of 5th Internnational Symposium on Practical Aspects of Declarative Languages (PADL). (2003) 110–127

A Free, Defined Variables and Substitution

Expressions

$$
\begin{aligned}
fv(\mathbf{null}) &= \bullet \\
fv(\mathbf{throw}) &= \bullet \\
fv(\mathsf{x}) &= \{\mathsf{x}\} \\
fv(e.\mathsf{f}) &= fv(e) \\
fv(e.\mathsf{m}(e_\star^{1..n})) &= fv(e) \cup \bigcup_{1..n} fv(e_\star) \\
fv(C(e_\star^{1..n}) &= \bigcup_{1..n} fv(e_\star) \\
fv(a?\{\mathsf{x}\colon C \Rightarrow e\}/\{d\}) &= fv(a) \cup fv(d) \cup fv(e) \backslash \{\mathsf{x}\} \\
fv(e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..k}\}) &= \bigcup_{1..n} fv(e_\star) \cup \bigcup_{1..k} fv(c_\star)
\end{aligned}
$$

Case Clauses and Patterns

$$
\begin{aligned}
fv(\mathbf{case}\ p_\star^{1..n} \Rightarrow b) &= fv(p_\star^{1..n}) \cup fv(b) \backslash \bigcup_{1..n} dv(p_\star) \\
fv(\mathsf{x}) &= \bullet \\
fv(\hat{v}.\mathsf{m}(p_\star^{1..n}) &= fv(\hat{v})
\end{aligned}
$$

Expressions

$$
\begin{aligned}
dv(\mathbf{null}) &= \bullet \\
dv(\mathbf{throw}) &= \bullet \\
dv(\mathsf{x}) &= \bullet \\
dv(e.\mathsf{f}) &= dv(e) \\
dv(e.\mathsf{m}(e_\star^{1..n})) &= dv(e) \cup \bigcup_{1..n} dv(e_\star) \\
dv(C(e_\star^{1..n}) &= \bigcup_{1..n} dv(e_\star) \\
dv(a?\{\mathsf{x}\colon C \Rightarrow e\}/\{d\}) &= dv(a) \cup dv(d) \cup dv(e) \cup \{\mathsf{x}\}
\end{aligned}
$$

Case Clauses and Patterns

$$
\begin{aligned}
dv(\mathbf{case}\ p_\star^{1..n} \Rightarrow b) &= \bigcup_{1..n} dv(p_\star) \cup dv(b) \\
dv(\mathsf{x}) &= \{\mathsf{x}\} \\
dv(\hat{v}.\mathsf{m}(p_\star^{1..n}) &= \bigcup_{1..n} dv(p_\star)
\end{aligned}
$$

Contexts

$$
\begin{aligned}
fv([]) &= \bullet \\
dv([]) &= \bullet
\end{aligned}
$$

Substitution in expressions σ :

$$\begin{aligned}
\mathsf{x}\sigma &= \begin{cases} e & \text{if } \langle \mathsf{x}, e\rangle \in \sigma \\ \mathsf{x} & \text{otherwise} \end{cases} \\
e.\mathsf{f}\sigma &= e\sigma.\mathsf{f} \\
e.\mathsf{m}(e_\star^{1..n})\sigma &= e\sigma.\mathsf{m}(e_\star\sigma^{1..n}) \\
\mathbf{null}\sigma &= \mathbf{null} \\
C(e_\star^{1..n})\sigma &= C(e_\star\sigma^{1..n}) \\
\mathbf{throw}\sigma &= \mathbf{throw} \\
a?\{\mathsf{x}\colon C \Rightarrow b\}/\{d\}\sigma &= a\sigma?\{\mathsf{y}\colon C \Rightarrow b\sigma\}/\{d\sigma\} \\
e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..k}\}\sigma &= e_\star\sigma^{1..n}\ \mathbf{match}\ \{c_\star\sigma^{1..k}\} \\
\\
(\mathbf{case}\ p_\star^{1..n} \Rightarrow b)\sigma &= \mathbf{case}\ p_\star\overline{\sigma}^{1..n} \Rightarrow b\sigma
\end{aligned}$$

Substitution in patterns $\overline{\sigma}$:

$$\begin{aligned}
\mathsf{x}\overline{\sigma} &= \mathsf{x} \\
\hat{v}.\mathsf{m}(p_\star^{1..k})\overline{\sigma} &= \hat{v}\sigma.\mathsf{m}(p_\star\overline{\sigma}^{1..k})
\end{aligned}$$

B Proofs for Type Soundness

Lemma 1 (Uniqueness) *For all a, if $a \Downarrow q$ then for all q', if $a \Downarrow q'$ then $q = q'$.*

Proof By induction on $a \Downarrow q$ and case analysis on q'. □

Lemma 2 (Termination) *For all a and all q, it holds that if $a \Downarrow q$ then $a \not\Uparrow$.*

Proof By induction on $a \Downarrow q$ and inversion of $a \Uparrow$. We only show (Rfld).

Case $a \equiv e.\mathsf{f}$ (Rfld), (Dfld) Assume $a \Downarrow q$, then by (Rfld) $e \Downarrow v$. By i.h. $e \not\Uparrow$, so (Dfld) is not available. Hence $e.\mathsf{f} \not\Uparrow$. □

Lemma 3 (Subtypes have all Fields) *If $C{<:}D$, $C \neq \mathsf{Exc}$ then $\mathsf{fields}(C) = \mathsf{fields}(D); \mathsf{g}_\star : E_\star^{1..m}$.*

Proof By induction on the derivation of $C{<:}D$.

Case (Sobj) Then $\mathsf{fields}(\mathsf{Obj}) = \bullet$

Case (Sthr) Cannot happen

Case (Sref) Trivial

Case (Sext) Then the definition of fields is applied

Case (Stran) The i.h. is applied twice. □

Lemma 4 (Subtypes have all Methods) *If $C{<:}D$, $C \neq \mathsf{Exc}$ and $\mathsf{mtype}(\mathsf{m}, D) = an(C_\star^{1..n})B$, then $\mathsf{mtype}(\mathsf{m}, C) = an(C_\star^{1..n})B$.*

Proof By induction on the derivation of $\mathsf{mtype}(\mathsf{m}, D) = an(C_\star^{1..n})B$ and case analysis over $C{<:}D$.

Case (Sobj) Cannot happen, since Obj has no methods.

Case (Sthr) Cannot happen

Case (Sref) Trivial

Case (Sext) If C does not contain a definition for m, then the definition of mtype is applied. Otherwise, class definition of C is well-typed, thus $\mathsf{override}(\ an(C_\star^{1..n})B, \mathsf{m}, D)$ asserts that $\mathsf{mtype}(\mathsf{m}, C) = \mathsf{mtype}(\mathsf{m}, D)$.

Case (Stran) The i.h. is applied twice □

Lemma 5 (Subtypes yield Refined Environment)
If $C{<:}D$ and $\Gamma; D \ni p \dashv \Gamma'$ then $\Gamma; C \ni p \dashv \Gamma''$ for some $\Gamma''{<:}\Gamma'$.

Proof By induction on $\Gamma; D \ni p \dashv \Gamma'$.

Case (TPvar) Then $\Gamma; D \ni \mathsf{x} \dashv \{\mathsf{x}\colon D\}$ and we can also derive $\Gamma; C \ni \mathsf{x} \dashv \{\mathsf{x}\colon C\}$. From $C{<:}D$ follows $\{\mathsf{x}\colon C\}{<:}\{\mathsf{x}\colon D\}$

Case (TPextr)

Then $\Gamma; D \ni \hat{v}.\mathsf{m}(p_\star^{1..n}) \dashv {\Gamma'}_\star^{1..n}$ and subpatterns have derivations $\Gamma; D_\star \ni p_\star \dashv {\Gamma'}_\star^{1..n}$ for some casefield types $D_\star^{1..n}$. The expected type is not used for typing the subpatterns, thus the subderivations can be reused as is, yielding $\Gamma; C \ni \hat{v}.\mathsf{m}(p_\star^{1..n}) \dashv {\Gamma'}_\star^{1..n}$. □

Lemma 6 (Refined Environment preserves Typing)
If $C_\star{<:}D_\star^{1..n}$ and $\Gamma, \mathsf{x}_\star : D_\star^{1..n} \vdash e \in B$ then $\Gamma, \mathsf{x}_\star : C_\star^{1..n} \vdash e \in A$ for $A{<:}B$.

Proof By straightforward induction on $\Gamma, \mathsf{x}_\star : D_\star^{1..n} \vdash e \in B$. □

Lemma 7 (Weakening) *If $\Gamma \vdash d \in S$ and $\mathsf{x} \notin fv(d)$, then $\Gamma, \mathsf{x}\colon T \vdash d \in S$ for any T.*

Proof Straightforward induction on $\Gamma \vdash d \in D$. □

Lemma 8 (Substitution Lemma) *If $\Gamma, \mathsf{x}_\star : B_\star^{1..n} \vdash b \in D$ and $\bullet \vdash \dot{u}_\star \in A_\star^{1..n}$ for $A_\star{<:}B_\star^{1..n}$, $\dot{u}_\star \in \mathsf{Values} \cup \{\mathbf{null}\}$ then $\Gamma \vdash b\,\{\mathsf{x}_\star \mapsto \dot{u}_\star^{1..n}\} \in C$, for $C{<:}D$.*

Proof By induction on the derivation of $\Gamma, \mathsf{x}_\star : B_\star^{1..m} \vdash b \in D$. Let $\sigma = \{\mathsf{x}_\star \mapsto \dot{u}_\star^{1..n}\}$ and $\Gamma' = \Gamma, \mathsf{x}_\star : B_\star^{1..m}$.

Case (Tvar) $b \equiv \mathsf{x}$

i) If $\mathsf{x} = \mathsf{x}_i$ for some i, then $\mathsf{x}\sigma = \dot{u}_i$ with $\Gamma \vdash \dot{u}_i \in A_i$ and $A_i{<:}B_i$ by assumption, **(Weakening)**.
ii) Otherwise, $\mathsf{x}\sigma = \mathsf{x}$ and rule (Tvar).

Case (Tthr),(Tnul) trivial because $b\,\sigma \equiv b$

Case (Tfld) $b \equiv e.\mathsf{f}$ We have $\Gamma' \vdash e \in E$ and i.h. yields $\Gamma \vdash e\sigma \in E'$ for $E'{<:}E$. By **(Subtypes have all Fields)** we have $\mathsf{fields}(E') = \mathsf{fields}(E); \mathsf{g}_\star : D_\star^{1..m}$ and (Tfld) finishes the case.

Case (Tinvk) $b \equiv e.\mathsf{m}(e_\star^{1..n})$ We have $\Gamma' \vdash e \in E$ and $\mathsf{mtype}(\mathsf{m}, E) = an(C_\star^{1..n})D$. The i.h. yields $\Gamma \vdash e\sigma \in E'$ for $E'{<:}E$. We also have $\Gamma' \vdash e_\star \in E_\star^{1..n}$ for $E_\star{<:}C_\star^{1..n}$ and i.h. yields $\Gamma \vdash e_\star\sigma \in {E'}_\star^{1..n}$ for $E'_\star{<:}E_\star^{1..n}$. By **(Subtypes have all Methods)**, we know $\mathsf{mtype}(\mathsf{m}, E') = \mathsf{mtype}(\mathsf{m}, E)$. Transitivity of $<:$ and rule (Tinvk) finishes the case.

Case (Tnew) $b \equiv C(e_\star^{1..n})$ We have $\mathsf{fields}(C) = C_\star^{1..n}$ and $\Gamma' \vdash e_\star \in E_\star^{1..n}$ with $E_\star{<:}C_\star^{1..n}$. The i.h. yields $\Gamma \vdash e_\star\sigma \in {E'}_\star^{1..n}$ for $E'_\star{<:}E_\star^{1..n}$. Transitivity of $<:$ and (Tnew) finish the case.

Case (Ttst) $b \equiv \dot{u}?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\}$ We have $\Gamma' \vdash \dot{u} \in A$, $\Gamma' \vdash d \in E_1$, and $\Gamma' \vdash e \in E_0$ and $E_\star{<:}D^{0,1}$. The i.h. yields $\Gamma \vdash \dot{u}\sigma \in A'$, $\Gamma \vdash d\sigma \in E'_1$, and $\Gamma \vdash e\sigma \in E'_0$ with $A'{<:}A$, $E'_1{<:}E_1$ and $E'_0{<:}E_0$. Transitivity of $<:$ and (Ttst) finishes the case.

Case (Tmch) $b \equiv e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\}$ We have $\Gamma' \vdash e_\star \in C_\star^{1..n}$ and i.h. yields $\Gamma \vdash e_\star\sigma \in {C'}_\star^{1..n}$ for $C'_\star{<:}C_\star^{1..n}$.

For each $j \in 1..m$, let $c_j \equiv \mathbf{case}\ p_\star^{1..n} \Rightarrow b_j$ (we omit the extra j index for patterns). We have a case typing $\Gamma'; C_\star^{1..m} \vdash c_j \in D_j$ via $\Gamma'; C_\star \ni p_\star \dashv \Gamma''^{1..n}_\star$ and $\Gamma', \Gamma''^{\,1..n}_\star \vdash b \in D_j$.

By **(Subtypes yield Refined Environment)**, we get $\Gamma'; C'_\star \ni p_\star \dashv \Gamma'''^{\,1..n}_\star$ for $\Gamma'''_\star{<:}\Gamma''^{\,1..n}_\star$.
By **(Refined Environment preserves Typing)** we get $\Gamma, \Gamma'''^{1..n}_\star \vdash b_j \in D'_j$ for $D'_j{<:}D_j$.
Applying the i.h. yields $\Gamma, \Gamma'''_\star n \vdash b_j\sigma \in D''_j$ for $D''_j{<:}D'_j$.
For the combined lubs, we have $\bigsqcup D''^{1..m}_\star{<:}\bigsqcup {D_\star}^{1..m}$, and rule (Tmch) finishes the case.

□

Lemma 9 (Preservation)
If $a \Downarrow q$ and $\bullet \vdash a \in C$, then $\bullet \vdash q \in C'$ for some $C'{<:}C$.

Proof For $q \equiv \mathbf{throw}$ and $q \equiv \mathbf{null}$, rules (Tthr) and (Tnul) yield the proof. Otherwise, induction on $a \Downarrow v$.

Case (Rfld) $a \equiv e.\mathsf{f}_i$
The premises of (Tfld) are $\bullet \vdash e \in C_0$ and $\mathsf{fields}(C_0) = \mathsf{f}_\star\colon C_\star^{1..m}$ with $C = C_i$.
We have $e \Downarrow D(\dot{w}_\star^{1..n})$ and $v = w_i$.
By i.h. $\bullet \vdash D(\dot{w}_\star^{1..n}) \in D$ for $D{<:}C_0$.
By **(Subtypes have all Fields)**, we obtain $m \leq n$ and $f_i \in \mathsf{fields}(D)$.
Finally, from $\bullet \vdash D(\dot{w}_\star^{1..n}) \in D$ and rule (Tnew) we know $\bullet \vdash \dot{w}_i \in E_i$ with $E_i{<:}C_i$.

Case (Rinvk) $a \equiv e.\mathsf{m}(e_\star^{1..n})$
The premises of (Tinvk) are $\Gamma \vdash e \in E$, $\mathsf{mtype}(\mathsf{m}, E) = an(C_\star^{1..n})C_0$, $\bullet \vdash e_\star \in C_\star^{1..n}$.
Then $e \Downarrow D(\dot{w}_\star^{1..m})$, $e_\star \Downarrow \dot{v}_\star^{1..n}$, $\mathsf{mbody}(\mathsf{m}, D) = (\mathsf{x}_\star^{1..n})e_0$ with $\bullet \vdash e_0 \in C$. Under substitution $\sigma = \{\mathsf{this} \mapsto D(\dot{w}_\star^{1..m}), \mathsf{x}_\star \mapsto \dot{v}_\star^{1..n}\}$, the body evaluates as $e_0\,\sigma \Downarrow \dot{v}$.
Applying the i.h. for the receiver yields $D{<:}E$.
By **(Subtypes have all Methods)** we get $\mathsf{mtype}(\mathsf{m}, D) = \mathsf{mtype}(\mathsf{m}, E)$.
Applying the i.h. for the arguments yields $\bullet \vdash \dot{v}_\star \in {C'}_\star^{1..n}$ for $C'_\star{<:}C_\star^{1..n}$.
By **(Substitution Lemma)** we get $\bullet \vdash e_0\sigma \in C'$ for $C'{<:}C$.
Applying the i.h. for the body then yields $\bullet \vdash \dot{v} \in C''$ for $C''{<:}C'$. Transitivity of subtyping finishes the case.

Case (Rnew) $a \equiv D(\dot{w}_\star^{1..n})$ then $a \Downarrow a$, and $D{<:}D$ by (Sref).

Case (Rcst) $a \equiv e?\{\mathsf{x}\colon C \Rightarrow b\}/\{d\}$
We have $e \Downarrow D(\dot{w}_\star^{1..n})$, $D{<:}C$ and $b\,\{\mathsf{x} \mapsto D(\dot{w}_\star^{1..n})\} \Downarrow v$.

Using typing premises from (Ttst), we apply the **(Substitution Lemma)** and then the i.h.

Case (Rskp) $a \equiv e?\{\mathsf{x}\colon C \Rightarrow b\}/\{d\}$

We have $e \Downarrow D(\dot{w}_\star^{1..n})$, $D \not<:C$ and $d \Downarrow v$.

Using typing premises from (Ttst), we apply the i.h. to d, yielding $\bullet \vdash d \in C'$.

Case (Rmch) $a \equiv e_\star^{1..m}\ \mathbf{match}\ \{c_\star^{1..l}\}$. Let i be the index of the matching case.

The premises of (Tmch) include $\bullet \vdash e_\star \in {C_\star}^{1..m}$ and case typing $\bullet; {C_\star}^{1..m} \vdash c_i \in D_i$ for $D_i <: C$.

The case typing has premises $\bullet; C_\star \ni p_\star \dashv {\Gamma'_\star}^{1..n}$ and $\bullet; {\Gamma'}^{1..m} \vdash b \in D_i$ where $c_i \equiv \mathbf{case}\ p_\star \Rightarrow b$.

We have $e_\star \Downarrow {\dot{v}_\star}^{1..m}$ as well as $\dot{v}_\star \curvearrowright p_\star \dashv {\sigma_\star}^{1..m}$ and $b\,{\sigma_\star}^{1..m} \Downarrow \dot{v}$.

Applying the i.h. to $e_\star$ yields $\bullet \vdash \dot{v}_\star \in {C'_\star}^{1..n}$ for ${C'_\star} <: {C_\star}^{1..n}$.

By **(Substitution Lemma)**, $\bullet \vdash b\sigma_\star \in D'_i$ with $D'_i <: D_i$.

Applying the i.h. yields $\bullet \vdash \dot{v} \in D''_i$ with $D''_i <: D'_i$.

This yields the desired type $C' = D''_i <: \bigsqcup {D_\star}^{1..m} = C$, by transitivity of $<:$ and properties of the least upper bound operator $\sqcup$. □

Lemma 10 (Progress)
If $\bullet \vdash a \in C$ and $a \not\Downarrow q$ for all q, then $a \Uparrow$.

Proof By coinduction and case analysis over a. We recall the principle of coinduction: In order to show that $R = \{a \mid \text{for all } q\ .\ a \not\Downarrow q \text{ and } \bullet \vdash a \in C\}$ is included in $\Uparrow$, it suffices to show that R is preserved by the divergence rules. This means, if we replaced each assertion $a \Uparrow$ with $a \in R$ and assumed that the conclusion holds, we have to be able to show that the premises hold as well. To do this, we need proofs for $e \not\Downarrow$ for the subexpressions e of a. These can be obtained from inversion of "blocked" evaluation rules.

Case $a \in \{\mathbf{throw}, \mathbf{null}\}$ and $a \equiv \mathsf{x}$ are not interesting, since $a \notin R$

Case $a \equiv e_0.\mathsf{f}$ and (Rfld), (Cfld) are blocked. By $\bullet \vdash a \in C$ and (Tfld), we also have $\bullet \vdash e_0 \in C_0$. Thus, either

i) $e_0 \not\Downarrow q_0$ for any q_0. This amounts to $e_0 \in R$ and shows that (Dfld) preserves R.
ii) $e_0 \Downarrow \mathbf{throw}$ or $e_0 \Downarrow \mathbf{null}$, but this contradicts (Cfld) blocked.
iii) $e_0 \Downarrow D(\dot{w}_\star^{1..n})$ but by **(Preservation)** and **(Subtypes have all fields)**, this contradicts (Rfld) blocked.

Case $a \equiv e_0.\mathsf{m}(e_\star^{1..n})$ and (Rinvk),(Crcv) and (Carg) are blocked. By $\bullet \vdash a \in C$ and (Tinvk), we also have $\bullet \vdash e_0 \in C_0, \bullet \vdash e_\star \in {C_\star}^{1..n}$, $\mathsf{mtype}(\mathsf{m}, C_0) = ({D_\star}^{1..n})$ and ${C_\star} <: {D_\star}^{1..n}$.

Thus, either

i) $e_0 \not\Downarrow q_0$ for any q_0. This amounts to $e_0 \in R$ and shows that (Drcv) preserves R.
ii) $e_0 \Downarrow \mathbf{throw}$ or $e_0 \Downarrow \mathbf{null}$ but this contradicts (Crcv) blocked.
iii) $e_0 \Downarrow D(w_\star^{1..n})$. By **(Preservation)**, $D <: C_0$ and by **(Subtypes have all methods)**, $\mathsf{mbody}(\mathsf{m}, D) = ({\mathsf{x}_\star}^{1..n})b$. We can distinguish further
   a) There exists i with $e_\star \Downarrow {\dot{v}_\star}^{1..i-1}$ and $e_i \not\Downarrow q_0$ for any q_0. Then $e_i \in R$ and (Darg) preserves R.
   b) There exists i with $e_\star \Downarrow {\dot{v}_\star}^{1..i-1}$ and $e_i \Downarrow \mathbf{throw}$, but this contradicts (Carg) blocked

c) $e_\star \Downarrow \dot{v}_\star^{1..n}$ and **(Preservation)** yields $\bullet \vdash v_\star \in D_\star^{1..n}$ for $D_\star{<:}C_\star^{1..n}$. Then let $\sigma = \{\mathsf{this} \mapsto D(w_\star^{1..m})\ \mathsf{x}_\star \mapsto v_\star^{1..n}\}$ and consider $b\sigma$. Either
   1. $b\,\sigma \not\Downarrow q$ for any q. This amounts to $b\,\sigma \in R$ and shows that (Dinvk) preserves R.
   2. $b\,\sigma \Downarrow q$, but this contradicts (Rinvk) blocked.

Case $a \equiv C(e_\star^{1..n})$ and (Rnew), (Cnew) are blocked. By $\bullet \vdash a \in C$ and (Tnew), we also have $\bullet \vdash e_\star \in A_\star^{1..n}$, $\mathsf{fields}(C) = B_\star^{1..n}$ and $A_\star{<:}B_\star^{1..n}$. Thus, either

i) there exists i with $e_\star \Downarrow v_\star^{1..i-1}$ and $e_i \not\Downarrow q$ for any q. Then $e_i \in R$ and (Dnew) preserves R
ii) there exists i with $e_\star \Downarrow v_\star^{1..i-1}$ and $e_i \Downarrow \mathbf{throw}$, but this contradicts (Cnew) blocked.
iii) $e_\star \Downarrow v_\star^{1..n}$, but this contradicts (Rnew) blocked.

Case $a \equiv e?\{\mathsf{x}\colon C \Rightarrow b\}/\{d\}$ and (Rcst),(Rskp),(Ctst) are blocked. By $\bullet \vdash a \in C$ and (Ttst), we have all premises of the rule (Ttst). Thus either

i) $e \not\Downarrow q$ for any q, then $e \in R$ and (Dtst) preserves R.
ii) $e \Downarrow \mathbf{throw}$, but this contradicts (Ctst) blocked.
iii) $e \Downarrow D(\dot{v}_\star^{1..n})$. There are several subcases to consider:
   a) $D{<:}C$, and for $\sigma = \{\mathsf{x} \mapsto D(\dot{v}_\star^{1..n})\}$, $b\,\sigma \not\Downarrow q$ for any q. Then $b\,\sigma \in R$ and (Dcst) preserves R.
   b) $D{<:}C$, and for $\sigma = \{\mathsf{x} \mapsto D(\dot{v}_\star^{1..n})\}$, $b\,\sigma \not\Downarrow \mathbf{throw}$ but this contradicts (Rcst) blocked.
   c) $D \not<:C$, and $d \not\Downarrow q$ for any q. Then $d \in R$ and (Dskp) preserves R.

Case $a \equiv e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\}$ and (Rmch),(Cmch) are blocked.

By $\bullet \vdash a \in C$ and (Tmch), we have $\bullet \vdash e_\star \in A_\star^{1..n}$, for all j a case typing $\bullet; A_\star^{1..n} \vdash c_j \in D_j$, and for each body b_j a typing $\Gamma'_j \vdash b_j \in D_j$.
Thus, either

i) there exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \not\Downarrow q$ for any q. Then $e_i \in R$ and (Dmch) preserves R.
ii) there exists i with $e_\star \Downarrow \dot{v}_\star^{1..i-1}$ and $e_i \Downarrow \mathbf{throw}$ or $e_i \Downarrow \mathbf{null}$, but this contradicts (Cmch) blocked.
iii) $e_\star \Downarrow \dot{v}_\star^{1..n}$. Then we distinguish these cases:
   a) if all cases reject, this contradicts that the last case always accepts.
   b) There exists an i such that $\forall j < i\ .\ v_\star^{1..n}; c_j \Downarrow \mathbf{reject}$, $c_i = \mathbf{case}\ p_\star^{1..n} \Rightarrow b$ and $\dot{v}_\star \curvearrowright p_\star \dashv \sigma_\star^{1..n}$. Then, either
      1) $b\,\sigma_\star^{1..n} \not\Downarrow q$ for any q, then $b\,\sigma_\star^{1..n} \in R$ and (Dbdy), (Dcase) preserve R
      2) $b\,\sigma_\star^{1..n} \Downarrow q$, which contradicts (Rmch) blocked.

Thus, R is preserved by all divergence rules, so $R \subseteq \Uparrow$.

□

Theorem 1 (Type Soundness)
If $\bullet \vdash a \in C$ then either $a \Uparrow$ or $a \Downarrow q$ for some q with $\bullet \vdash q \in C'$, $C'{<:}C$.

Proof Consequence of **(Progress)** and **(Termination)**.

C Proofs for Correctness

Definition 1 (Equivalence) *For d, e expressions with $fv(d) = fv(e) = \emptyset$, d is equivalent to e (written $d \approx e$), if both of these conditions hold: 1. for all q, if $d \Downarrow q$ then $e \Downarrow q$, and 2. if $d \Uparrow$ then $e \Uparrow$.*

Definition 2 (Open Equivalence) *For expressions d, e with $fv(d) \cup fv(e) \subseteq X$, d is open-equivalent to e (written $X \Vdash d \approx e$) if $d\sigma \approx e\sigma$ for all substitutions σ with $X \subseteq \mathsf{dom}(\sigma)$.*

Definition 3 (Target Context) *A target context is defined by the following grammar:*

$$\begin{aligned}\xi, \zeta ::= {} & [\,] \mid \xi.\mathsf{f} \mid \xi.\mathsf{m}(b_\star^{1..n}) \mid a.\mathsf{m}(b_\star, \xi, b_\star^{1..]i[..n}) \\ & \mid \xi?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\} \mid a?\{\mathsf{x}\colon C \Rightarrow \xi\}/\{e\} \mid a?\{\mathsf{x}\colon C \Rightarrow d\}/\{\xi\}\end{aligned}$$

Lemma 11 (Substitution preserves Equivalence) *If $X \Vdash d \approx e$, then for any substitution σ with $\mathsf{dom}(\sigma) \subseteq X$, it holds that $X \backslash \mathsf{dom}(\sigma) \Vdash d\sigma \approx e\sigma$.*

Proof Let $X \Vdash d \approx e$ and σ be a substitution. We have to show that for any substitution ρ with $\mathsf{dom}(\rho) = X \backslash \mathsf{dom}(\sigma)$, it holds that $(d\sigma)\rho \approx (e\sigma)\rho$. Considering that substitution is associative, this amounts to $d(\sigma\rho) \approx e(\sigma\rho)$. We observe that $\sigma\rho$ is a substitution with $X = \mathsf{dom}(\sigma\rho)$, and the equivalence follows from $X \Vdash d \approx e$. □

The following definition can be used to show that a rewrite rule takes a typed expression to another typed expression.

Definition 4 (Target Context Typing) *A context is assigned a type $[C]D$ according to the following typing rule.*

$$\frac{\Gamma, \mathsf{z}\colon C \vdash \xi[\mathsf{z}] \in D \quad \mathsf{z}\ \textit{fresh}}{\Gamma \vdash \xi \in [C]D}$$

By the **(Substitution Lemma)** it is easy to see, that if $\Gamma \vdash \xi \in [C]D$ and $\Gamma \vdash a \in B$ for $B{<:}C$, then $\Gamma \vdash \xi[a] \in D'$ for $D'{<:}D$.

Theorem 2 (Congruence) *If $X \Vdash d \approx e$, then $Y \Vdash \xi[d] \approx \xi[e]$ for $Y = fv(\xi[d]) \cup fv(\xi[e])$.*

Proof By induction on ξ. For each context shape, we consider possible terminating and divergent computations under a value substitution σ.

Case $\xi \equiv [\,]$ then $Y = X$ and $X \Vdash d \approx e$ by assumption.

Case $\xi \equiv \zeta.\mathsf{f}$

i) $(\zeta[d].\mathsf{f})\sigma \Downarrow \mathbf{throw}$ by (Cfld). Then, we have $\zeta[d]\sigma \Downarrow \mathbf{throw}$ or $\zeta[d]\sigma \Downarrow \mathbf{null}$. By i.h., we obtain $\zeta[e]\sigma \Downarrow \mathbf{throw}$ (resp. $\zeta[e]\sigma \Downarrow \mathbf{null}$) and $(\zeta[e].\mathsf{f})\sigma \Downarrow \mathbf{throw}$ by (Cfld).

ii) $(\zeta[d].\mathsf{f})\sigma \Downarrow w$ by rule (Rfld). Then, we have $\zeta[d]\sigma \Downarrow C(\dot{v}_\star^{1..n})$ and $\mathsf{f}\colon D \in \mathsf{fields}(C)$. By the i.h., we obtain $\zeta[e]\sigma \Downarrow C(\dot{v}_\star^{1..n})$ and $(\zeta[e].\mathsf{f})\sigma \Downarrow w$ by (Rfld).

iii) $(\zeta[d].\mathsf{f})\sigma \Uparrow$ by (Dfld). Then $\zeta[d]\sigma \Uparrow$, by i.h. $\zeta[e]\sigma \Uparrow$ and $(\zeta[e].\mathsf{f})\sigma \Uparrow$ by (Dfld)

Case $\xi \equiv \zeta.\mathsf{m}(b_\star^{1..n})$

i) $(\zeta[d].\mathsf{m}(b_\star^{1..n}))\sigma \Downarrow \mathbf{throw}$ by (Crcv). Then $\zeta[d]\sigma \Downarrow \mathbf{throw}$ or $\zeta[d]\sigma \Downarrow \mathbf{null}$. By i.h. $\zeta[e]\sigma \Downarrow \mathbf{throw}$ (resp. $\zeta[e]\sigma \Downarrow \mathbf{null}$) and $(\zeta[e].\mathsf{m}(b_\star^{1..n}))\sigma \Downarrow \mathbf{throw}$ by (Crcv).

ii) $(\zeta[d].\mathsf{m}(b_\star^{1..n}))\sigma \Downarrow \mathbf{throw}$ by (Carg). Then $b_i\,\sigma \Downarrow \mathbf{throw}$ or $b_i\,\sigma \Downarrow \mathbf{null}$ and $\zeta[e].\mathsf{m}(b_\star^{1..n}) \Downarrow \mathbf{throw}$ by (Carg).

iii) $(\zeta[d].\mathsf{m}(b_\star^{1..n}))\sigma \Downarrow q$ by (Rinvk). Then $\zeta[d]\sigma \Downarrow C(\dot{v}_\star^{1..m})$. By i.h., $\zeta[e]\sigma \Downarrow C(\dot{v}_\star^{1..m})$ and by (Rinvk), $\zeta[e].\mathsf{m}(b_\star^{1..n}) \Downarrow q$.

Case $\xi \equiv a.\mathsf{m}(b_\star, \zeta, b_\star^{1..]i[..n})$

This case is similar to the preceding case.

Case $\xi \equiv \zeta?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\}$

i) $(\zeta[d]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow \mathbf{throw}$ by (Ctst). Then $\zeta[d]\sigma \Downarrow \mathbf{throw}$. By i.h. $\zeta[e]\sigma \Downarrow \mathbf{throw}$ and $(\zeta[e]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow \mathbf{throw}$ by (Ctst).

ii) $(\zeta[d]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow q$ by (Rcst). Then $\zeta[d]\sigma \Downarrow E_0(\dot{v}_\star^{1..n})$ for some $E_0{<:}E$ and $a\sigma\{\mathsf{x} \mapsto E_0(\dot{v}_\star^{1..n})\} \Downarrow q$. The i.h. yields $(\zeta[e])\sigma \Downarrow E_0(\dot{v}_\star^{1..n})$ and $(\zeta[e]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow q$ by (Rcst).

iii) $(\zeta[d]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow q$ by (Rskp). Then we either have $\zeta[d]\sigma \Downarrow \mathbf{null}$ or $\zeta[d]\sigma \Downarrow C(\dot{v}_\star^{1..n})$ for some $C \not<:E$, and $b\sigma \Downarrow q$. The i.h. yields $\zeta[e]\sigma \Downarrow \mathbf{null}$ resp. $\zeta[e]\sigma \Downarrow C(\dot{v}_\star^{1..n})$ and $(\zeta[e]?\{\mathsf{x}\colon E \Rightarrow a\}/\{b\})\sigma \Downarrow q$ by (Rskp).

Case $\xi \equiv a?\{\mathsf{x}\colon E \Rightarrow \zeta\}/\{b\}$

This case is similar to the preceding case, with i.h. applied with $\sigma\{\mathsf{x} \mapsto w\}$ for (Rcst).

Case $\xi \equiv a?\{\mathsf{x}\colon E \Rightarrow b\}/\{\zeta\}$

This case is similar to the preceding case.

□

Definition 5 (Derived Form for Value Definition) *The expression form* $\mathbf{val}\,\mathsf{x}\colon C = a; d$ *abbreviates* $a?\{\mathsf{x}\colon C \Rightarrow d\}/\{d\{\mathsf{x} \mapsto \mathbf{null}\}\}$ *and is typed according to the scheme*

$$\frac{\Gamma \vdash a \in A \qquad \Gamma, \mathsf{x}\colon A \vdash d \in D \qquad \Gamma \vdash d\{\mathsf{x} \mapsto \mathbf{null}\} \in D'}{\Gamma \vdash a?\{\mathsf{x}\colon C \Rightarrow d\}/\{d\{x \mapsto \mathbf{null}\}\} \in D}\ (\mathsf{Ttst})$$

With **(Substitution Lemma)**, it is easy to see that $D'{<:}D$ and thus $D \sqcup D' = D$

Lemma 12 (ValDef Equivalences) *Let* $b \equiv \mathbf{val}\,\mathsf{x}\colon A = a; d$ *where* $\Gamma \vdash a \in A$ *and* σ *some substitution that agrees with* Γ*. Then*

I.	If $a\sigma \Downarrow w$	then $b\,\sigma \approx d\{\mathsf{x} \mapsto w\}\sigma$.
II.	If $a\sigma \Downarrow \mathbf{null}$	then $b\,\sigma \approx d\{\mathsf{x} \mapsto \mathbf{null}\}\sigma$.
III.	If $a\sigma \Downarrow \mathbf{throw}$	then $b\,\sigma \Downarrow \mathbf{throw}$.
IV.	If $a\sigma \Uparrow$	then $b\,\sigma \Uparrow$.

Proof (Sketch) We note that by **(Substitution Lemma)** and **(Preservation)**, the type test cannot fail except for **null**. With this observation, I. follows from (Rcst), II. from (Rskp), III. from (Ctst) and IV. from (Drcv). □

Definition 6 (Translation) *For* $a \equiv e_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\}$ *with* $\Gamma \vdash a \in C$*, the translation* $[\![\Gamma \vdash a \in C]\!]$ *is defined as the application of a suitable rule in Fig. 7.*

Lemma 13 (Split) *Let* $a \equiv \mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{c_\star^{1..m}\}$ *with* $\Gamma \vdash a \in A$,
with $c_1 = \mathbf{case}\ \mathsf{x}_\star^{1..i-1}\ \hat{v}.\mathsf{m}(\pi_\star^{1..k})\ p_\star^{i+1..n} \Rightarrow b$,
and $\mathsf{xtype}(\Gamma, \hat{v}, \mathsf{m}) = @\mathbf{safe}(\mathsf{Obj})C$.
For any substitution σ it holds that :

I. $\hat{v}.\mathsf{m}(\mathsf{z}_i)\sigma \Downarrow w$ implies $a\sigma \approx (\mathsf{z}_\star, \mathsf{z}_i,\ \mathsf{w}_\star^{1..k}, \mathsf{z}_\star^{1..]i[..n}\ \mathbf{match}\ \{expand_{\hat{v}.m}(c_\star^{1..m})\})\sigma$
II. $\hat{v}.\mathsf{m}(\mathsf{z}_i)\sigma \Downarrow \mathbf{null}$ implies $a\sigma \approx (\mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{other_{\hat{v}.m}(c_\star^{1..m})\})\sigma$

Proof *I*. Let $a' \equiv (\mathsf{z}_\star, \mathsf{z}_i,\ \mathsf{w}_\star^{1..k}, \mathsf{z}_\star^{1..]i[..n}\ \mathbf{match}\ \{expand_{\hat{v}.m}(c_\star^{1..m})\})\sigma$.
Terminating computation $a\sigma \Downarrow q$ can only happen through (Rmch).
We show pattern acceptance and rejection coincides in a and a'.
Let $\mathsf{z}_\star\sigma^{1..n}; c_j\sigma \Downarrow \mathbf{reject}$. Then acceptance as well as rejection was established to the left of column i, in column i, or to the right of column i of the original match in a.
Patterns to the left of i are not changed by *expand*. Patterns to the right of i are merely moved to index $i + k$, but test the same input values.

- If in clause c_j a pattern in column i is of the form $\hat{v}.\mathsf{m}(\pi_\star'^{1..k})$ for some $\pi_\star'^{1..k}$, the function $expand$ lifts patterns $\pi_\star'^{1..k}$ appear in c_j', to be matched against $w_\star^{1..k}$.
  The same derivations for acceptance and rejection can be reused: whenever acceptance by (mextr) is derived with $w_\star \curvearrowright \pi_\star' \dashv \rho_\star^{1..k}$ the corresponding $\rho_\star^{1..k}$ are also obtained in c_j'. Moreover, whenever rejection is derived through (rchild), then $w_h \curvearrowright \pi_h' \dashv \mathbf{reject}$ for some h can be also be derived in c_j'. The additional variable pattern at position i does not affect the outcome, since it was chosen fresh.
- If in clause c_j, a pattern with a different extractor appears in column i, then the outcome is obviously the same. The additional variable patterns in columns $i+1..i+k$ act as dummy patterns for discarded input values $w_\star^{1..k}$.

Divergent computation $a\sigma \Uparrow$ is only derivable with (Dcase), and (Dbdy). By the same reasoning as above, patterns in c_j' have the same acceptance/rejection behavior as those in c_j. So the matching case c_i in (Dbdy) produces the same substitution ρ that makes the body diverge.

Proof *II*. Let $a' \equiv (\mathsf{z}_\star^{1..n}\ \mathbf{match}\ \{other_{\hat{v}.m}(c_\star^{1..m})\})\sigma$.
The hypothesis is enough to derive pattern rejection by (rnull). It is clear that this rejection judgment for column i causes all c_j with the same $\hat{v}.\mathsf{m}(\pi_\star'^{1..k})$ in column i to reject input values $\mathsf{z}_\star^{1..n}\sigma$. For a formal proof, this is not enough, so we argue that we can actually produce a proof of case rejection for every such c_j.
To this end, we have to look at all patterns to the left of column i. It is important to note that for each such pattern, we can derive either an acceptance or a rejection judgment: First of all, the syntax allows only extractor patterns $C(\hat{v}_\star^{1..l}).\mathsf{m}(\ldots)$, so no null dereferencing can occur. Furthermore, **by condition "@safe" divergent patterns and exceptions in patterns are ruled out**. An inductive argument is then applied to each extractor pattern in column $l < i$ to derive either an acceptance or a rejection judgement: Every extractor call of terminates normally (guaranteed by $@\mathbf{safe}$), yielding (so we stop with a rejection judgment) or a value w, whose case-fields are used for the remaining subpatterns (with the induction hypothesis yielding acceptance or rejection of subpatterns, which is the used to derive acceptance or rejection

for the current pattern). So every pattern in column l on the top-level will either accept or reject z_l.

Now, a sequence of acceptance judgments followed by a rejection judgment in input z_l for $l < i$ yields the desired case rejection for c_j. If instead all patterns to the left of colum i accept, then we use rejection of $\hat{v}.\mathsf{m}(\pi'^{1..k}_\star)$ to derive rejection of the whole case clause.

Putting it all together, we conclude that omitting these case clauses will not alter the behavior of a. The expression a' uses $other$ to omit exactly these cases, and will contain only those cases that still need to be tested for acceptance or rejection (with at least one case always accepting, by convention). Thus, every evaluation involving (Rmch) (which involves one crucial pattern acceptance judgment) for a can be simulated by a corresponding evaluation involving (Rmch) for a'. □

Theorem 3 (Correctness of $[\![\,]\!]$) *For $a \equiv a_\star^{1..n}$ $\mathbf{match}$ $\{c_\star^{1..m}\}$, $fv(a) = X$, typing $\Gamma \vdash a \in A$, translation $a' = [\![\Gamma \vdash a \in A]\!]$ it holds that $X \Vdash a \approx a'$.*

Proof By case distinction on $[\![\Gamma \vdash a \in C]\!]$. Let σ be a substitution that closes a.

Case (Tmp) Let σ be a substitution with $X \subseteq \mathsf{dom}(\sigma)$. Then **(ValDef Equivalences)** yields the desired result.

Case (Var) $a = \mathsf{z}_\star^{1..n}$ $\mathbf{match}$ $\{c_\star^{1..m}\}$ and $a' = b\{\mathsf{x}_\star \mapsto \mathsf{z}_\star^{1..n}\}$
where $c_1 = \mathbf{case}\ \mathsf{x}_\star^{1..n} \Rightarrow b$

Since $\mathsf{z}_\star\sigma \in \mathsf{Values} \cup \{\mathbf{null}\}$, we can ignore (Cmch),(Dmch). By (mcase) and (mvar), the first row matches, yielding $\rho = \{\mathsf{x} \mapsto \mathsf{z}\sigma\}$. We then exploit that $b\,\rho \equiv b\{\mathsf{x}_\star \mapsto \mathsf{z}_\star^{1..n}\}\sigma$

i) $a\sigma \Downarrow q$ by (Rmch). Then $b\rho \Downarrow q$ thus $b\{\mathsf{x}_\star \mapsto \mathsf{z}_\star^{1..n}\}\sigma \Downarrow q$
ii) $a\sigma \Uparrow$ by (Dcase), (Dbdy). Then $b\rho \Uparrow$ thus $b\{\mathsf{x}_\star \mapsto \mathsf{z}_\star^{1..n}\}\sigma \Uparrow$

Case (Mix) $a = \mathsf{z}_\star^{1..n}$ $\mathbf{match}$ $\{c_\star^{1..m}\}$ and $\hat{v}.\mathsf{m}(\mathsf{z}_i)?\{\mathsf{x}\colon B \Rightarrow d\}/\{e\}$
where $c_1 = \mathbf{case}\ \mathsf{x}_\star^{1..i-1}\hat{v}.\mathsf{m}(p_\star^{1..k})\ p_\star^{i+1..n} \Rightarrow b$ and d, e as described in Figure 7.

Since $\mathsf{z}_\star\sigma \in \mathsf{Values} \cup \{\mathbf{null}\}$, we can ignore (Cmch),(Dmch). We distinguish two cases.

- $\hat{v}.\mathsf{m}(\mathsf{z}_i)\sigma \Downarrow w$. Then by **(Preservation)**, (Rcst), we get $a'\sigma \approx d\sigma$. We finish the case with Lemma **(Split)** and transitity of $\approx$.
- $\hat{v}.\mathsf{m}(\mathsf{z}_i)\sigma \Downarrow \mathbf{null}$. Then by (Rskp), $a'\sigma \approx e\sigma$. We finish the case with Lemma **(Split)** and transitity of $\approx$.

□

D The complete *transform* function

$$
\begin{array}{ll}
\mathit{transform}(\mathbf{null}) & = \mathbf{null} \\
\mathit{transform}(\mathsf{x}) & = \mathsf{x} \\
\mathit{transform}(e.\mathsf{f}) & = \mathit{transform}(e).\mathsf{f} \\
\mathit{transform}(e.\mathsf{m}({e_\star}^{1..n})) & = \mathit{transform}(e).\mathsf{m}({e'_\star}^{1..n}) \\
 & \qquad \text{where } e'_\star = \mathit{transform}(e_\star)^{1..n} \\
\mathit{transform}(\mathbf{throw}) & = \mathbf{throw} \\
\mathit{transform}(a?\{\mathsf{x}\colon C \Rightarrow d\}/\{e\}) & = a'?\{\mathsf{x}\colon C \Rightarrow d'\}/\{e'\} \\
 & \qquad \text{where } d' = \mathit{transform}(d) \\
 & \qquad \text{and } e' = \mathit{transform}(e)
\end{array}
$$

$$
\mathit{transform}({e_\star}^{1..n}\ \mathbf{match}\ \{{c_\star}^{1..k}\}) = \mathit{transform}(\mathit{rewrite}({e_\star}^{1..n}\ \mathbf{match}\ \{{c_\star}^{1..k}\}))
$$

$$
\begin{array}{ll}
\mathit{transform}(\mathbf{def}\ \mathsf{m}(\mathsf{x}_\star : {C_\star}^{1..n})\colon C = \{e\}) & = \mathbf{def}\ \mathsf{m}(\mathsf{x}_\star : {C_\star}^{1..n})\colon C = \{\mathit{transform}(e)\} \\
\mathit{transform}(\mathbf{class}\ C(\mathsf{x}_\star : {C_\star}^{1..n}) \lhd D\ \{{an}_\star\ {md}_\star^{1..k}\}) & = \mathbf{class}\ C(\mathsf{x}_\star : {C_\star}^{1..n}) \lhd D\ \{{an}_\star\ {md'}_\star^{1..k}\} \\
 & \qquad \text{where } md' = \mathit{transform}(md) \\
\mathit{transform}({cd}_\star^{1..n}; e) & = \mathit{transform}({cd}_\star)^{1..n}; \mathit{transform}(e)
\end{array}
$$

E A Simplistic Approximation of Safe Methods

Although we do not treat safety in detail, we mention the fact that sometimes the following fragment of the calculus may sometimes be acceptable as a sublanguage for safe pattern matching. A method can be proven @**safe**, if its body is in the restricted set of "safe expressions".

Safe Expressions

$$
\frac{\mathsf{x} \in \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathsf{x}\ \text{not null}} \qquad \frac{\Gamma, \mathsf{x}\colon C \vdash b\ \text{not null} \quad \Gamma \vdash d\ \text{not null}}{a?\{\mathsf{x}\colon C \Rightarrow b\}/\{e\}\ \text{not null}}
$$

$$
\frac{}{\Gamma \vdash \mathbf{null}\ \mathsf{safe}} \qquad \frac{\Gamma \vdash e\ \text{not null} \quad \Gamma \vdash e\ \mathsf{safe}}{\Gamma \vdash e.\mathsf{f}\ \mathsf{safe}} \qquad \frac{\Gamma \vdash a\ \mathsf{safe} \quad \Gamma, \mathsf{x}\colon C \vdash b\ \mathsf{safe} \quad \Gamma \vdash e\ \mathsf{safe}}{a?\{\mathsf{x}\colon C \Rightarrow b\}/\{e\}\ \mathsf{safe}}
$$

F A Formal Characterization of @safe

For completeness, we give a formal definition of what the @**safe** modifier expresses.

Definition 7 *A method in a class C that is* @**safe** *satisfies the following property: For any sequence of optional values ${\dot v_\star}^{1..m}$ and any arguments ${\dot w_\star}^{1..n}$ such that $e = C({v_\star}^{1..m}).\mathsf{m}({\dot w_\star}^{1..n})$ is a well-typed expression, there exists an optional value $\dot w$ with a derivation of $e \Downarrow \dot w$.*