Safe Type-level Abstraction in Scala

Adriaan Moors * Frank Piessens
K.U. Leuven
{adriaan, frank}@cs.kuleuven.be

Martin Odersky
EPFL
martin.odersky@epfl.ch

Abstract

Most formal accounts of object-oriented languages have focussed on type soundness: the safety that type checking provides with respect to *term-level* computation and abstractions. However, with type-level abstraction mechanisms becoming increasingly more sophisticated, bringing this guarantee to the level of types has become quite pressing. We call this property *kind soundness*: kind checking ensures that type constructors are never applied to unexpected type arguments. We present Scalina, a purely object-oriented calculus that employs the same abstraction mechanisms at the type level as well as at the kind level. Soundness for both levels can thus be proven by essentially the same arguments. Kind soundness finally allows designers of type-level abstractions to join their term-level colleagues in relying on the compiler to catch deficiencies before they are discovered by their clients.

1. Introduction

Scalina is a purely object-oriented calculus that provides the formal underpinning for our implementation of higher-kinded types [19] in Scala [20]. Scalina introduces a number of novelties with respect to earlier object-oriented calculi [16, 21, 10]. The most notable improvement over the νObj calculus is that kind checking ensures type applications never "go wrong": we dub this property *kind soundness*.

Traditionally, most object-oriented languages and the underlying formalisms use a mix of FP-style and OO-style abstraction. The former style is based on lambda abstraction and function application, and OO-style abstractions are built using abstract members and composition (via subclassing or mixin composition).

Java, for example, uses functional abstraction for methods and classes, which may be parametric in types and values. Of course, Java also supports OO-style abstraction: a class with an abstract method abstracts from the implementation of that method. A subclass is expected to provide the concrete implementation.

Like νObj, Scalina is a purely object-oriented calculus: there are no constructs for parameterisation. Yet, as we will demonstrate, Scalina is able to express the same abstractions as, for example, System F_{ω}^{sub}[8, 23, 9], with the same safety guarantees.

* The first author is supported by a grant from the Flemish IWT. Part of the reported work was performed during a 3-month stay at EPFL.

FOOL '08 13 January, San Francisco, California, USA.
ACM .

Listing 1. Expressing `Iterable` using parameterisation

```
trait Iterable[A, Container[X]] {
  def map[B](f: A ⇒ B): Container[B]
}

trait List[A] extends Iterable[A, List]
```

The rest of this section elaborates on the problem statement and gives some initial insight into our solution. Then, we get our feet wet with Scalina's syntax and intuitions in Section 2, before delving deeper in the levels of terms (Section 3) and types (Section 4). The latter two sections discuss computation and classification at the respective levels. We briefly motivate Scalina's design and position it in the design space in Section 5. In Section 6 we make the relation between Scalina and System F_{ω}^{sub} more precise. We sketch the meta-theory in Section 7. Finally, we briefly discuss related work (Section 8) before concluding in Section 9.

1.1 Kind soundness

Scala supports two styles of abstraction: the functional style uses parameterisation, whereas abstract members represent the object-oriented way. It is natural to ask whether one style can be used exclusively. At first sight, the object-oriented style can encode the functional one. We restrict the discussion to the technicalities of the encoding; the impact on the programming experience is outside the scope of this paper.

Scala's abstract type members closely correspond to type parameters, and abstract type member refinement can be seen as the object-oriented counterpart of type application. Abstract type member refinement is a restricted form of mixin composition that can be used to override abstract type members with concrete ones. However, it turns that out this encoding does not preserve the safety properties that are ensured by parameterisation.

To make this concrete, Listing 1 uses parameterisation to express the well-known `Iterable` abstraction in Scala. The `Iterable` trait (an abstract class) takes two type parameters: the first one represents the type of the elements, and the second one abstracts over the type constructor of the container. To denote that it abstracts over a type constructor, the `Container` parameter declares a formal type parameter `X`.

Listing 2 demonstrates the object-oriented style. Here, `Iterable` abstracts over the type of its elements and the container using abstract members. The `A` type member is inherited from `TypeFunction1`, and the `Container` type constructor parameter is represented as an abstract type member that is bounded to be a `TypeFunction1`. `map`'s result type is expressed by refining `Container`'s abstract type member `A` so that it equals `B`.

So far, the encoding remained faithful to the original. However, a discrepancy emerges when we encode an erroneous program. The type application `Iterable[A, NumericList]` in Listing 3 is not

Listing 2. Encoding `Iterable`'s type parameters as members

```
trait TypeFunction1 { type A }

trait Iterable extends TypeFunction1 {
  type Container <: TypeFunction1

  def map[B](f: A ⇒ B): Container{type A = B}
}

trait List extends Iterable { type Container = List }
```

Listing 3. `NumericList`: an illegal subclass of `Iterable`

```
trait NumericList[A <: Number]
    extends Iterable[A, NumericList]
```

Listing 4. The encoding of `NumericList` eludes the type checker

```
trait NumericList extends Iterable {
  type A <: Number
  type Container = NumericList // Incorrect, but no
    error reported!
}
```

allowed by the compiler, whereas we will see its encoding is accepted without warning. If it were not ruled out, `map`'s result type could apply any type `B` to `NumericList`, while it accepts only subtypes of `Number`. By ruling out `Iterable[A, NumericList]`, the compiler prevents this error from ever happening.

Unfortunately, the encoding does not preserve this property, which we call "kind soundness". This is illustrated by Listing 4, which is considered a valid Scala program. The compiler silently accepts this program, even though we could never complete its implementation (at some point we will have to instantiate a `NumericList` for an arbitrary type of elements, and the compiler will catch our mistake). To relate this to type soundness, the value-level equivalent of this oversight would be to allow passing a function of type, e.g., `Number ⇒ Any` to a function that expects a `Any ⇒ Any`.

Note that this indulgence does not imply *type* unsoundness, as these erroneous types cannot be instantiated. Nonetheless, we regard it as a shortcoming of the compiler that these vacuous intersection types are allowed to slip by unnoticed. Even though they are prevented from being instantiated, they could be unmasked earlier.

To motivate this desire for early detection of these inconsistencies, consider the analogy with abstract classes. Suppose classes would be allowed to be abstract implicitly, so that accidental abstract classes would not be discovered until a client attempts to instantiate them. However, this situation is considered undesirable by most languages, so that an abstract class must be marked as such explicitly. This eliminates the possibility that the programmer simply forgot to implement a method.

Not detecting erroneous type applications, which manifest themselves as intersection types that unexpectedly do not have any instances, has the same effect as allowing any class to be abstract implicitly: the error is detected eventually, but it could have been signalled earlier. Even though other uses of intersection types might sensibly result in empty types, we do not consider this to be one of them.

This kind unsoundness has its roots in the νObj calculus [21], which allows abstract type members to be refined *covariantly*, thus

Listing 5. Using un-members to recover kind soundness

```
trait TypeFunction1 { deferred type A }

trait Iterable extends TypeFunction1 {
  type Container <: TypeFunction1

  def map[B](f: A ⇒ B): Container◁{type A = B}
}

trait List extends Iterable { type Container = List }

trait NumericList extends Iterable {
  deferred type A <: Number // error: covariant change
     not allowed
  type Container = NumericList
}
```

`NumericList <: TypeFunction1`, so that the encoding of the erroneous type application results in a valid program.

We recover early error detection in Scalina by differentiating covariant and contravariant members, instead of assuming they all behave covariantly. This distinction corresponds to the fact that some members abstract over input, whereas others represent the output of the abstraction. Input members should behave contravariantly, like the types of function arguments, whereas covariance is required for output members, which correspond to a function's result type. With this distinction, a purely object-oriented calculus can encode functional-style abstraction with the same safety guarantees.

If we look at the problem from the point of view of the *clients* of an abstraction, we distinguish external and internal clients. External clients supply information to an abstraction without knowing exactly which subtype of the abstraction they are dealing with. Therefore, the constraints on these missing pieces of information must only be *weakened* in subtypes. Internal clients, which are tightly related by subtyping, should be able to strengthen the result of the abstraction.

Thus, Scalina complements Scala's covariant type members with contravariant ones, which we shall call "un-members". Listing 5 shows a pseudo-Scala rendition of the encoding, where un-members are indicated using the **`deferred`** keyword. They are made concrete by external clients using the `... ◁{ ... }` construct.

Since the `A` type member is an input to the abstraction, it must behave contravariantly, so that `NumericList` is not allowed to strengthen the bound on the `A` un-member that it inherited from `Iterable`.

1.2 Methodology

To summarise the above example, a programmer should use un-members to model the input to an abstraction. This corresponds to the arguments of a method or the type parameters of a generic class. Normal members are used to define the result of the abstraction.

Note that un-members and abstract members impose an ordering discipline. A type un-member that classifies a value un-member must be refined before the value can be supplied. This corresponds to a polymorphic value in functional programming. Furthermore, types may contain abstract members, but objects must not. Therefore, an object cannot be created until the abstract members have been made concrete.

The example in Section 2.3 will illustrate these points in more detail.

1.3 Contributions

Functional abstraction clearly distinguishes a function's arguments from its result. In the object-oriented setting, a similar distinction

$t,\ u$	$::=$		term
	$\mid$	x	variable
	$\mid$	$t\,.\,l$	selection
	$\mid$	$t \lhd \{\, cm \,\}$	refinement
	$\mid$	$\mathbf{new}\ T$	instantiation
TT	$::=$		type
	$\mid$	$TT \,\#\, L$	selection
	$\mid$	$TT \lhd \{\, cm \,\}$	refinement
	$\mid$	$\{\, x \,[\![:\ TT \,]\!] \Rightarrow \overline{m_i}^{\,i} \,\}$	structural type
	$\mid$	$TT_1 \,\&\, TT_2$	intersection type
	$\mid$	$p\,.\,\mathbf{type}$	singleton type
	$\mid$	$\mathbf{Any}$	top type
	$\mid$	$\mathbf{Nothing}$	bottom type
	$\mid$	$\Box\ TT$	necessarily TT
$T,\ S$	$::=$		type or un-type
	$\mid$	TT	type
	$\mid$	$\mathbf{Un}\,[\ TT\ ]$	un-type
m	$::=$		member
	$\mid$	$\mathbf{val}\, l\ :\ T \,[\![= \ t \,]\!]$	value member
	$\mid$	$\mathbf{type}\, L\ :\ K \,[\![= \ T \,]\!]$	type member
cm	$::=$		refinement member
	$\mid$	$\mathbf{val}\, l\ =\ t$	value member
	$\mid$	$\mathbf{type}\, L\ =\ T$	type member

Figure 1. Scalina Syntax (terms and types)

must be made for abstract members. We introduce un-members, which safely model the input to an abstraction, and re-use traditional members to represent the result of the abstraction. Thus, an object with un-members may be thought of as a curried function that takes its keyword arguments in any order. The members of such an object represent its results.

We study purely object-oriented abstraction in a dependently typed, three-level calculus that uses the same concepts for abstraction and computation on terms and types. As in the νObj calculus, function application is decomposed into refinement and member selection. Because the level of types is modelled after the level of terms, a type-level function is modelled as a type with type un-members.

The distinction between un-members, which behave contravariantly, and normal, covariant, members, is instrumental in proving soundness on the level of types and kinds. Due to the symmetric design of our calculus, the soundness proofs proceed by similar arguments at both levels.

2. Scalina: Syntax and Intuitions

Scalina is a three-level object-oriented calculus: we distinguish terms (objects), types, and kinds. Terms are for computation, types are used for classification as well as computation, and the role of kinds is strictly limited to classification. Computation is performed using two mechanisms: member selection and member refinement. Classification is more intricate, ranging from merely structural descriptions of the classified entities over nominal classification, the intersection of classifiers, singletons, and strictly empty classifiers.

2.1 Syntax

Figures 1 and 2 outline Scalina's syntax. We use '$[\![\ldots]\!]$' to denote the optionality of '$\ldots$'.

The term level consists of member selection, member refinement, and instantiation. Analogously, a type may be a type selection, a refinement or a structural type. A structural type binds the self variable x in the members it includes; if the type of the self variable is not specified, it is assumed to be the structural type itself. We use the meta-variable R to refer to a structural type. Additionally, a type may be an intersection type, a singleton type (that depends on a path), the top or the bottom of the subtype lattice, or an un-type. Finally, we introduce $\Box T$, which stands for the result of refining all of T's un-members with unknown terms and types. We will discuss this construct in more detail in Section 2.2.3.

Figure 2 defines the shape of kinds, paths, values, and the typing context Γ. A path is a chain of member selections that starts with a variable or an instantiation expression `new T`, which represents an object. We mainly restrict the shape of paths to simplify the proofs in the meta-theory.

2.2 Core concepts

Before describing the rules that define computation and classification in Scalina, we build up intuitions about the core concepts that underlie these mechanisms.

2.2.1 Members and un-members

Members are the liaisons between the different levels: a type describes the value members that may be selected on the terms it classifies, as well as the type members that may be selected on the type itself. The description of a member consists of the label of the member, the classifier of the entity it stands for and – if the member is concrete – the actual entity it is bound to (its right-hand side, or RHS). For value members, the classifier is a type and the RHS is a term, and type members specify the kind that classifies the type they are bound to.

Scalina's *un-members* are a more radical departure from Scala. Un-members are used to encode parameterisation: they are placeholders for members that must be provided by the client of the abstraction, much like the arguments of a function. Un-members are turned into normal members using member refinement, which corresponds to passing arguments to a function. An entity with multiple un-members is the equivalent of a curried function: refining one of the un-members results in an entity with one less un-member to be refined. Once all un-members have been refined, the member representing the function's result may be selected to complete the application. This constitutes the essence of computation – on terms as well as types – in Scalina.

Members and un-members can be seen as the two halves of the contract specified by a classifier: members are *available* to the client, whereas it must *supply* the un-members. Note that abstract members have different semantics from un-members: an abstract member is made concrete using composition within a subtyping hierarchy, while an un-member is to be supplied by an external client. A type with abstract members cannot be instantiated. An abstract type can however be constrained (using the kind `Concrete (R)`) so that it does not contain any abstract members.

2.2.2 Terms

The canonical form of a term is an object. For syntactic economy, and since Scalina does not model effects yet, an object is represented by the instantiation of a type without abstract members. Conceptually, an entity is just a vessel for denoting to which entity each of its members – as described by the entity's classifier – is bound. Thus, an object contains mappings (from a label to a term) for all of the members specified in its type. Operationally, un-members can be thought of as members that are simply absent from this mapping.

2.2.3 Types

> If, on the term level, parameterising over functions is useful, doing the same on the level of types sounds like an obvious thing to do.
>
> Erik Meijer

KK	::=		kind
	|	$\mathbf{In}\,(\,T_1\,,\,T_2\,)$	interval kind
	|	$\mathbf{Struct}\,(\,R\,)$	structural kind
	|	$\mathbf{Nominal}\,(\,R\,)$	nominal kind
	|	$\mathbf{Concrete}\,(\,R\,)$	concrete kind
K	::=		kind or un-kind
	|	KK	kind
	|	$\mathbf{Un}\,[\,KK\,]$	un-kind
p	::=		path
	|	x	variable
	|	$p\,.\,l$	selection
	|	$\mathbf{new}\ T$	instantiation
v	::=		value
	|	$\mathbf{new}\ T$	instantiation
Γ	::=		typing context
	|	$\emptyset$	
	|	$\Gamma\,,\,x\,:\,T$	assume x has type T

Figure 2. Scalina Syntax (kinds, etc.)

To generalise Meijer's motivation for higher-kinded types [18], rephrasing in our terminology: "If, on the term level, abstracting over terms that themselves abstract over terms is useful, doing the same on the level of types sounds like an obvious thing to do." Scalina manifestly supports this view by using the same abstraction mechanism on both levels: entities that abstract over other entities (using un-members) are themselves first-class entities.

Types play a dual role: besides computation, their main purpose is classifying terms. As explained in the introduction, types differ from terms in that they may contain abstract members for abstraction towards subtyping clients. Another distinction with the term level is that we intend to tone down type-level computation so that it becomes decidable (this is future work).

Types classify terms by specifying the labels and the types of the members that may be selected on these terms. A structural type classifies all terms that have the prescribed members. Note that we use kinds to distinguish nominal types from structural ones. An intersection type is inhabited by the terms that inhabit both its constituent types. A singleton type classifies exactly one object and an un-type does not classify any terms at all. An un-type is used as the classifier of a value un-member.

Type-level computation uses the same concepts as computation at the term level. However, because types may contain abstract members, we must be more careful. For soundness, type member selection is only allowed on types that (eventually) consist solely of concrete members, although the exact RHS need not be known. Type selection on a singleton type is always safe, even if the selected type member's right-hand side is not known statically. As long as it is not an un-member, the object that the singleton type depends on, could not have been created unless that member was concrete.

In Scala, these abstract type members may *only* be selected on singleton types. Scalina generalises this to the notion of *concrete types*, so that abstract type members may be selected on any type that necessarily contains only concrete type members, which naturally includes singleton types.

Similarly, it is always safe to assume that the type of the self variable does not contain any un-members: the self-variable can only be accessed as a consequence of an external member selection, which in turn is not allowed on objects with un-members. To exploit this invariant, we introduce the type $\Box T$, which stands for the result of refining all T's un-members. We shall illustrate this with an example in Section 2.3

The canonical form of a type is computed by performing all allowed member selections. This corresponds to the β-normal form in functional calculi.

2.2.4 Kinds

Kinds are only used for classifying types: they denote which members may be selected on the types they classify. An interval kind takes over the role of the bounds of a Scala-style abstract type member: `In(S, T)` is inhabited by types that are subtypes of `T` and supertypes of `S`.

`Struct(R)` is inhabited by types that have at least the members specified in `R`. These members must be well-formed under the assumption that the self variable has the declared self type. `Nominal(R)` is similar to `Struct(R)`, except that it serves as a marker for concrete type bindings that represent classes: normalisation should not replace a type selection of this kind with its right-hand side.

Finally, T has kind `Concrete(R)` if it has at least the members specified in `R`, and none of these are abstract. Furthermore, $\Box T$ must be a subtype of the self type declared in `R`, so that such a type may be instantiated (if it is not a singleton type) or be used as the target of type member selection.

2.3 Example: polymorphic lists

Listing 6 implements polymorphic lists with `map` to illustrate Scalina's support for parametric polymorphism and higher-order functions.

First, we introduce a little syntactic sugar.

- The kind $\star$ should be expanded to **`Struct`**`({x ⇒ })`,
- the type `p.L` is shorthand for `p.`**`type`**`#L`,
- the following type members are easily expanded:
  - **`type`** `L = R` becomes **`type`** `L :` **`Struct`**`(R) = R`,
  - **`type`** `L ≺ T` means **`type`** `L :` **`Nominal`**`(R) = T`, where `R` is the expansion of `T` to its least structural supertype (by the $\prec\!\!<$ relation defined in Fig. 7).

Since type members must always be nested in other types, our program is a term that instantiates the structural type that represents our "universe" (hence the `u` as the self variable). The type `u.`**`type`**`#Fun1`, or using syntactic sugar, `u.Fun1`, corresponds to a top-level class in Scala.

The first abstraction is a polymorphic unary function. `Fun1` is a nominal type that expands to a structural type with self variable `self`, whose type is assumed to be the nominal type itself, with all its un-members refined. This special self type is crucial: without it, the body of the function could not access its arguments, as these would be considered un-members. In this example, $\Box u.Fun1$ expands to the structural type {`x ⇒` **`type`** `T1: ⋆;` **`type`** `T2: ⋆;` **`val`** `v: x.T1;` **`val`** `apply: x.T2`}

`Fun1` takes two type arguments: the type of its value argument (`T1`) and the type of its result (`T2`). It also requires one value argument (`v`). These arguments are un-members, which must be provided by the caller of the function. The abstract `apply` member models the function's body. It must be made concrete before an actual function value can be created.

`List` abstracts over the type of its elements (`Element`) and declares one abstract method, `map`. We define a structural type, `map`, and an abstract value member with the same name. This way, it becomes more convenient to make this member concrete, subclasses of `List` may simply use an instance of the composition of `map` with another type that makes the apply method concrete.

The implementation of the `map` "method" in `Nil` simply returns a new instance of `Nil` with the appropriate element-type. In `Cons`, the result is another cons cell that applies the supplied function to the head of the list and that recurses on the tail.

Listing 6. Polymorphic List in Scalina

```
new { u ⇒
  type Fun1 ≺ {self : □ u.Fun1 ⇒
    type T1   : Un[⋆]
    type T2   : Un[⋆]
    val v     : Un[self.T1]
    val apply : self.T2
  }

  type List ≺ {self : □ u.List ⇒
    type Element : Un[⋆]
    type map = { selfMap : self.map ⇒
      type Tgt : Un[⋆]
      val fun: Un[u.Fun1◁{type T1=self.Element}
                    ◁{type T2=selfMap.Tgt}]

      val apply: u.List◁{type Element=selfMap.Tgt}
    }

    val map: self.map
  }

  type Nil ≺ u.List & {self : □ u.Nil ⇒
    val map : self.map =
      new self.map & { s : self.map ⇒
        val apply: u.List◁{type Element = s.Tgt}
          = new (u.Nil ◁{type Element = s.Tgt})
      }
  }

  type Cons ≺ u.List & {self : □ u.Cons ⇒
    val hd: self.Element
    val tl: u.List◁{type Element=self.Element}

    val map : self.map =
      new self.map & { s : □ self.map ⇒
        val apply: u.List◁{type Element=s.Tgt}
          = new u.Cons◁{type Element=s.Tgt} & {sc ⇒
              val hd: s.Tgt
                = (fun◁{val v=self.hd}).apply
              val tl: u.List◁{type Element=s.Tgt}
                = (self.tl.map
                     ◁{type Tgt=s.Tgt}
                     ◁{val fun=s.fun}).apply
            }
    }
  }
}
```

Note that `hd` and `tl` model constructor arguments: since they are required for an object of this type to be created, we use abstract members and not un-members.

Listing 7 shows a Scala rendition of the example that stays as close as possible to the Scalina version, using an idiomatic mix of functional and object-oriented abstractions.

Listing 7. Parametric List in Scala

```
abstract class List[Element] {
  def map[Tgt](fun: Element ⇒ Tgt): List[Tgt]
}

class Nil[Element] extends List[Element] {
  def map[Tgt](fun: Element ⇒ Tgt) = new Nil[Tgt]
}

abstract class Cons[Element] extends
                  List[Element] { self ⇒
  val hd: Element
  val tl: List[Element]

  def map[Tgt](fun: Element ⇒ Tgt) = new Cons[Tgt]{
    val hd: Tgt = fun(self.hd)
    val tl: List[Tgt] = self.tl.map[Tgt](fun)
  }
}
```

$\boxed{T \prec R}$ T expands to R at run time

$$\frac{T \ni L \mapsto S \setminus\setminus x \quad [x \mapsto T]\,S \prec R}{T \# L \prec R}\ \text{LU\_SEL}$$

$$\frac{T \prec \{x : \_ \Rightarrow m_1 .. m_n\} \quad \forall i \in 1..n.\ m'_i = \mathbf{refineIf}(m_i, cm)}{T \lhd \{cm\} \prec \{x : S \Rightarrow m'_1 .. m'_n\}}\ \text{LU\_RFN}$$

$$\frac{}{R \prec R}\ \text{LU\_REFL}$$

$$\frac{T \prec \{x \Rightarrow m_1 .. m_n\}}{T \prec \{x : \{x \Rightarrow m_1 .. m_n\} \Rightarrow m_1 .. m_n\}}\ \text{LU\_SELFX}$$

$$\frac{T_1 \prec \{x : S_1 \Rightarrow \overline{m_i}^{\,i}\} \quad T_2 \prec \{x : S_2 \Rightarrow \overline{m'_j}^{\,j}\} \quad \overline{m''_k}^{\,k} = \overline{m_i}^{\,i} \uplus \overline{m'_j}^{\,j}}{T_1 \,\&\, T_2 \prec \{x : S_2 \Rightarrow \overline{m''_k}^{\,k}\}}\ \text{LU\_MIX}$$

$$\frac{T \prec R}{(\mathbf{new}\ T).\,\mathbf{type} \prec R}\ \text{LU\_SING}$$

$$\frac{p.\,\mathbf{type} \ni l \mapsto p' \setminus\setminus x \quad ([x \mapsto p]\,p').\,\mathbf{type} \prec R}{p.\,l.\,\mathbf{type} \prec R}\ \text{LU\_SINGX}$$

Figure 3. Type Expansion for Run-time Lookup

3. Terms

3.1 Computation

Before we turn to the evaluation rules, we briefly consider how members are looked up at run time. For now, type members are statically bound and the role of types during evaluation is strictly limited to mapping the labels of the members of an object to terms. However, we anticipate support for virtual classes, which requires run-time lookup of types. In future work, we will prove that, for the current system, our approach to lookup is equivalent to statically expanding types that are instantiated to mappings of labels to terms, with the corresponding trivial run-time lookup function.

To look up a member at run time, we use the unfold relation ($\prec$) defined in Fig. 3, which relies on the following helper relations: $T \ni ll \mapsto e \setminus\setminus x$ denotes that T expands to a structural type that contains a member with label ll and right-hand side e in which the self variable x is bound. ll stands for either a term- or a type-label, and e is a term or a type. Similarly, $T \ni^{\text{un}} ll$ is derivable if T has an un-member with the specified label: it expects this un-member to be refined.

Furthermore, we factor out what it means to refine a single member: $\mathbf{refineIf}(m', cm)$ can be seen as a function that returns the refinement of the un-member m' with the cm's RHS if their respective labels are the same, otherwise it simply returns m'. Similarly, $m = \mathbf{refines}(m', cm)$ holds if m' and cm have the same label and m is the result of refining m' with cm. Finally, intersecting structural types corresponds to taking the union ($\uplus$) of

$\boxed{t \Rightarrow t'}$ t evaluates to t' in a single step

$$\frac{T \ni l \mapsto p \setminus\setminus x}{\mathbf{new}\, T\,.\, l \Rightarrow [\, x \mapsto \mathbf{new}\, T\,]\, p}\ \text{E\_SEL}$$

$$\frac{l \equiv \mathbf{label}\, cm \quad T \ni^{\mathsf{un}} l}{(\mathbf{new}\, T) \lhd\{cm\} \Rightarrow \mathbf{new}\,(T \lhd\{cm\})}\ \text{E\_RFN}$$

$$\frac{T \ni l \mapsto t \setminus\setminus x \quad [\, x \mapsto \mathbf{new}\, T\,]\, t \Rightarrow t' \quad U \equiv (\mathbf{new}\, T)\,.\, l\,.\,\mathbf{type}}{\mathbf{new}\, T \Rightarrow \mathbf{new}\,(T \,\&\, \{\_ \Rightarrow \mathbf{val}\, l : U = t'\})}\ \text{E\_CTXMEM}$$

$$\frac{t \Rightarrow u}{t' \lhd\{\mathbf{val}\, l = t\} \Rightarrow t' \lhd\{\mathbf{val}\, l = u\}}\ \text{E\_CTXRFNRHS}$$

$$\frac{t \Rightarrow u}{t \lhd\{cm\} \Rightarrow u \lhd\{cm\}}\ \text{E\_CTXRFNTGT}$$

$$\frac{t \Rightarrow u}{t\,.\, l \Rightarrow u\,.\, l}\ \text{E\_CTXSELTGT}$$

Figure 4. Term Evaluation

the corresponding sets of members, with concrete members in the right type overriding corresponding members in the left one.

The actual lookup proceeds by expanding a type to the corresponding structural type, after which looking up the required label is easy. The only tricky rule in the definition of the expansion relation $\prec$ is LU_SINGX. During evaluation, all types are of the shape `(new T).l1. ... .ln.type`. To reduce a selection $p.l$ to the base case, which is handled by LU_SING, we must lookup l in $p.type$ and inductively expand the resulting singleton type. To avoid extra complexity in the meta-theory, we factor in the evaluation rule for value selection instead of using evaluation directly.

The small-step evaluation relation that defines Scalina's operational semantics [31], is shown in Fig. 4. It consists of two evaluation rules and four congruence rules. The first evaluation rule, E_SEL, rewrites a member selection on an object to the RHS of that member after replacing the self variable by the object that was the target of the selection. The hypothesis that the label must be present in the object is represented as a lookup on the type. The side-condition that the member's RHS must be a path is crucial for proving type preservation: a path may only be replaced by a path. For now, all terms are paths in Scalina. However, in anticipation of adding effects to the calculus, we already distinguish paths and arbitrary terms.

E_RFN deals with refinement: it checks that the refined member was indeed an un-member (it was missing from the object), and then adds it to the object by refining the type that is used to track its members. The side-condition that l was an un-member is not necessary for proving type soundness, as the typing rules ensure that a well-typed term always meets it. We include it so that we can prove that un-members are never refined more than once by seeing that a program gets stuck if it violates that rule. However, by progress, well-typed terms never get stuck.

The only non-trivial congruence rule, E_CTXMEM, performs evaluation under member bindings, which can be thought of as running the constructor. This congruence rule is necessary to fulfil the side-condition of the rule for member selection. The shape of the type U is a technicality required by the proof of type preservation. It can be seen as an artefact of our using full-blown types for simply tracking the members of an object. The remaining congruence rules are standard.

$\boxed{\Gamma \vdash t : T}$ t has type T

$$\frac{\Gamma \vdash p : \{x \Rightarrow \mathbf{val}\, l : T\} \quad T \text{ not an un-type}}{\Gamma \vdash p\,.\, l : [\, x \mapsto p\,]\, T}\ \text{T\_SELPATH}$$

$$\frac{\Gamma \vdash t : \{x \Rightarrow \mathbf{val}\, l : T\} \quad x \notin \mathbf{FV}(T) \quad T \text{ not an un-type}}{\Gamma \vdash t\,.\, l : T}\ \text{T\_SEL}$$

$$\frac{\Gamma \vdash t : T \quad T \equiv \{x : S \Rightarrow m_1 \,..\, m_n\} \quad \exists i \in 1..n.\ m' = \mathbf{refines}(m_i, cm) \quad \Gamma, x : S \vdash m'\,\mathbf{WF} \quad x \notin \mathbf{FV}(cm)}{\Gamma \vdash t \lhd\{cm\} : T \lhd\{cm\}}\ \text{T\_RFN}$$

$$\frac{\Gamma \vdash T \prec\!\!\prec R \quad \Gamma \vdash R : \mathbf{Concrete}\,(R) \quad T \text{ not a singleton type} \quad T \text{ not of shape } \square T'}{\Gamma \vdash \mathbf{new}\, T : T}\ \text{T\_NEW}$$

$$\frac{\Gamma \vdash p : R}{\Gamma \vdash p : p\,.\,\mathbf{type}}\ \text{T\_SING}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T <: S}{\Gamma \vdash t : S}\ \text{T\_SUBSUME}$$

Figure 5. Term Classification

3.2 Classification

Figure 5 defines the shape of well-typed terms. When checking a value member selection, we treat the case where the target of the selection is a path (T_SELPATH) differently from when it is not (T_SEL). Suppose we treated both cases equally. Consider e.g., `t: {x ⇒ val a: x.b.type = x.b; val b: Any}`, so that `t.a : t.b.type`. Now, for the singleton type `t.b.type` to be well-formed, `t` must be a path. Therefore, the selection is not allowed if this is not the case. If the declared type of the member does not rely on the self variable, the target need not be a path.

Note that the rules for member selection rely on subsumption to discard all other members in the type of the target (as well as the selected member's RHS). This is not just a matter of cosmetics: this formulation ensures that the type of the target does not contain any other un-members, as they cannot be forgotten by subsumption. In terms of function types, the underlying intuition is that subsumption cannot change the number of arguments that a function takes. We will discuss this in more detail in the section on subtyping.

T_RFN classifies member refinement – in a sense, the dual of member selection. Essentially, this corresponds to checking the type of the argument while typing function application. This check is performed by requiring that there is a member with the same label as cm, and that the result of refining this member is well-formed.

We cannot use subsumption in this rule as the target that is being refined, may have several un-members. The type of refining a term is a refinement of the type of the term that is refined. Note that this type refinement could not be replaced by an intersection type. For such a type $T \& S$ to be well-kinded, S's members must conform to T's, but here, T contains an un-member whereas S does not, and subtyping can never relate un-members to regular members.

According to T_NEW, **new** T is well-typed with type T if T statically expands to the structural type R (by $\prec\!\!\prec$, defined in Fig. 7), where R is of kind **Concrete**(R). The remaining side

$\boxed{\Gamma \vdash T \rightsquigarrow T'}$ T normalises to T'

$$\frac{\Gamma \vdash T \ni \mathbf{type}\, L : K = S \setminus\setminus x \quad K \text{ not nominal} \quad \Gamma \vdash [x \mapsto T]\, S \rightsquigarrow S'}{\Gamma \vdash T\,\#\,L \rightsquigarrow S'} \text{ N\_SEL}$$

$$\frac{\Gamma \vdash T \rightsquigarrow \{x : S \Rightarrow m_1 .. m_n\} \quad \forall i \in 1..n.\ m_i' = \mathbf{refineIf}(m_i, cm)}{\Gamma \vdash T \triangleleft \{cm\} \rightsquigarrow \{x : S \Rightarrow m_1' .. m_n'\}} \text{ N\_RFN}$$

$$\frac{\Gamma \vdash T \rightsquigarrow \{x \Rightarrow m_1 .. m_n\}}{\Gamma \vdash T \rightsquigarrow \{x : \{x \Rightarrow m_1 .. m_n\} \Rightarrow m_1 .. m_n\}} \text{ N\_SELFX}$$

$$\frac{\Gamma \vdash T_1 \rightsquigarrow \{x : S_1 \Rightarrow \overline{m_i}^{\,i}\} \quad \Gamma \vdash T_2 \rightsquigarrow \{x : S_2 \Rightarrow \overline{m_j'}^{\,j}\} \quad \overline{m_k''}^{\,k} = \overline{m_i}^{\,i} \uplus \overline{m_j'}^{\,j}}{\Gamma \vdash T_1 \,\&\, T_2 \rightsquigarrow \{x : S_2 \Rightarrow \overline{m_k''}^{\,k}\}} \text{ N\_MIX}$$

$$\frac{\Gamma \vdash p : q.\mathbf{type}}{\Gamma \vdash p.\mathbf{type} \rightsquigarrow q.\mathbf{type}} \text{ N\_SNG}$$

Figure 6. Type Normalisation

conditions rule out degenerate cases. It is necessary to expand T to R and then check R has kind $\mathbf{Concrete}(R)$ because just checking that $T : \mathbf{Concrete}(R)$, implies that a *subset* R of T is safe to be instantiated, but not necessarily T itself.

Finally, a path has the corresponding singleton type if it is well-typed using the other rules (we assume finite derivations). Subsumption gives a well-typed term a less precise type.

4. Types and Kinds

4.1 Computation

Type normalisation, as shown in Fig. 6, is the "operational semantics" of the type level. To compute the normal form of a type, all allowed type member selections are performed, refinements and compositions of structural types are normalised to the corresponding structural type, and paths are safely rewritten if they are statically known to refer to the same object.

The selection of a type member with declared kind $\mathbf{Nominal}(R)$ is in normal form: these bindings must not be crossed. Hence the side-condition in N_SEL. If this condition were omitted, normalisation would no longer be kind-preserving, as a type of kind $\mathbf{Nominal}(R)$ would be replaced by a type of kind $\mathbf{Struct}(R)$, which is not a subkind of $\mathbf{Nominal}(R)$. By analogy to the term level, normalisation checks only the minimal side conditions, a separate theorem proves that it is kind-preserving.

Type expansion includes type normalisation, but is more aggressive: it replaces a nominal type binding with its (structural) right-hand side and widens singleton types. This is needed when calculating all the members in a type. Since type expansion must yield the least structural supertype of a type, we cannot use typing in the rules X_SING*, as this may invoke subsumption.

X_SINGVAR expands a singleton type that depends on a variable, that must therefore be in Γ. X_SINGNEW handles the other bases case, similar to run-time expansion of types. Finally, X_SINGSEL peels one layer of member selection from the path by approximating the outermost selection by its declared type.

X_NCSRY expands $\Box T$ to the expansion of T, after essentially stripping all the `Un[...]`'s from the declared types and kinds of its members. It achieves this by "pretending" to refine every un-member with an unknown right-hand side, so that un-members essentially become abstract members.

$\boxed{\Gamma \vdash T \prec\!\!\prec R}$ T expands to the structural type R

$$\frac{\Gamma \vdash T \ni \mathbf{type}\, L : K = S \setminus\setminus x \quad \Gamma \vdash [x \mapsto T]\, S \prec\!\!\prec R}{\Gamma \vdash T\,\#\,L \prec\!\!\prec R} \text{ X\_SEL}$$

$$\frac{\Gamma \vdash T \prec\!\!\prec \{x : S \Rightarrow m_1 .. m_n\} \quad \forall i \in 1..n.\ m_i' = \mathbf{refineIf}(m_i, cm)}{\Gamma \vdash T \triangleleft \{cm\} \prec\!\!\prec \{x : S \Rightarrow m_1' .. m_n'\}} \text{ X\_RFN}$$

$$\frac{}{\Gamma \vdash R \prec\!\!\prec R} \text{ X\_REFL}$$

$$\frac{\Gamma \vdash T \prec\!\!\prec \{x \Rightarrow m_1 .. m_n\}}{\Gamma \vdash T \prec\!\!\prec \{x : \{x \Rightarrow m_1 .. m_n\} \Rightarrow m_1 .. m_n\}} \text{ X\_SELFX}$$

$$\frac{\Gamma \vdash T_1 \prec\!\!\prec \{x : S_1 \Rightarrow \overline{m_i}^{\,i}\} \quad \Gamma \vdash T_2 \prec\!\!\prec \{x : S_2 \Rightarrow \overline{m_j'}^{\,j}\} \quad \overline{m_k''}^{\,k} = \overline{m_i}^{\,i} \uplus \overline{m_j'}^{\,j}}{\Gamma \vdash T_1 \,\&\, T_2 \prec\!\!\prec \{x : S_2 \Rightarrow \overline{m_k''}^{\,k}\}} \text{ X\_MIX}$$

$$\frac{x : T \in \Gamma \quad \Gamma \vdash T \prec\!\!\prec R}{\Gamma \vdash x.\mathbf{type} \prec\!\!\prec R} \text{ X\_SINGVAR}$$

$$\frac{\Gamma \vdash T \prec\!\!\prec R}{\Gamma \vdash (\mathbf{new}\, T).\mathbf{type} \prec\!\!\prec R} \text{ X\_SINGNEW}$$

$$\frac{\Gamma \vdash p.\mathbf{type} \ni \mathbf{val}\, l : T\_ \setminus\setminus x \quad \Gamma \vdash [x \mapsto p]\, T \prec\!\!\prec R}{\Gamma \vdash p.l.\mathbf{type} \prec\!\!\prec R} \text{ X\_SINGSEL}$$

$$\frac{\Gamma \vdash T \prec\!\!\prec \{x : S \Rightarrow m_1 .. m_h\} \quad \forall i \in 1..n.\ m_i' = \mathbf{refineIf}(m_i, \_)}{\Gamma \vdash \Box T \prec\!\!\prec \{x : S \Rightarrow m_1' .. m_n'\}} \text{ X\_NCSRY}$$

$\boxed{\Gamma \vdash T \ni m \setminus\setminus x}$

$$\frac{\Gamma \vdash T \prec\!\!\prec \{x : \_ \Rightarrow m_1 .. m_n\} \quad \exists i \in 1..n.\ m_i \equiv m}{\Gamma \vdash T \ni m \setminus\setminus x} \text{ X\_LU}$$

Figure 7. Type Expansion

4.2 Subtyping

Subtyping is mostly standard; the main novelties result from the interaction with un-members. Un-types introduce contravariance (by ST_UN), thus deviating from the norm of covariance. Since member subtyping is covariant, an un-member with declared type $\mathbf{Un}[T]$ may only be overridden by an un-member with a declared type that is a subtype of $\mathbf{Un}[T]$, thus it has the shape $\mathbf{Un}[T']$ with $T <: T'$. This means the overriding member weakens the restriction on the term that must be supplied by the client.

If a type S expands to a type T, then surely it is a subtype of that type T. Expanding S can be thought of as computing a least structural supertype of S, following type selections, crossing nominal type bindings and widening singleton types. Similarly, type equality ($\cong$) – the least reflexive, symmetric, and transitive relation that includes normalisation – is included (by ST_EQ).

The rules ST_ABS_UPPER and ST_ABS_LOWER incorporate the declared kinds of abstract type members into the subtyping relation.

For simplicity, the current version of Scalina does not model variance for type constructors, which explains why ST_INVAR considers type un-members to be invariant.

Besides the usual width- and depth-subtyping, subtyping of structural types must take extra care to never forget any un-members during subsumption. Intuitively, subsumption allows the client of a type to relax the expectations it has of that type, but it should not result in the client having fewer obligations.

Subtyping of members is defined in Fig. 10. Value members always behave covariantly; a type member becomes invariant as soon as it is made concrete. This is related to the fact that Scalina does not admit late-binding for type members.

To relate this to subtyping of function types in System F_{ω}^{sub}, a type of the shape $S \to T$ can only be a subtype of a type with the same shape, i.e., a function of the same arity. In our system, the number of un-members denotes the "arity" and types can only be subtypes if they have the same un-members. **`Any`** constitutes the only safe exception to this rule. It is safe for a structural type with un-members to be a subtype of **`Any`**, as no members can be selected on a term that is only known to have type **`Any`**.

Similarly, if subtyping forgets either constituent of an intersection type, any un-members in the forgotten type must still be present in the remaining one. For example, suppose we have a term of type {`x: S` ⇒ **`val`** `a: Un[T]; val b: T=x.a}` `&` {`x : S` ⇒ **`val`** `b : T`}, with `S` = {`x` ⇒ **`val`** `a: T; val b: T`}. If we were allowed to subsume the term's type to {`x : S` ⇒ **`val`** `b : T`}, we could access b before a had been refined.

For brevity, we use m **deferred** to check that m's classifier is of the shape **`Un`**`[T]` or **`Un`**`[K]`.

4.3 Classification

For the constructs that are shared by terms and types, classification is largely analogous. The main difference is that we have to be careful to only select types that will eventually become concrete. For objects, this is always the case, but types with abstract type members are still types. Whereas a term with type T is known to contain concrete versions of all members (not including un-members) in T, a type with kind $\mathbf{Struct}(R)$ may contain abstract members. Therefore, we introduce the kind $\mathbf{Concrete}(R)$ that classifies only types with only concrete members.

The kind of a structural type reflects the type members that may be selected on that type. To be well-kinded according to K_R, the members of a structural type must be well-formed under the assumption that the self variable has the declared self type. The well-formedness judgement for members is defined in Fig. 11.

The intersection of two structural types is classified by the kind that tracks the union of their members. Note that the self type of the overriding type (the right-most constituent) must be a subtype[1] of the type containing the overridden members. Each overriding member must be a submember of the corresponding member in T_1.

There are two ways for deriving that a set of members of a type are concrete. The easy way is if that type is a singleton type. Otherwise, for a type T to be classified as having a certain set of concrete members $m_1..m_n$, it must have a structural kind with declared self-type S and $\square T <: S$. Naturally, this structural kind must denote the $m_1..m_n$ as concrete. However, due to subsumption, this set of members may be a *subset* of the actual members of T. Nonetheless, any type member in R may safely be selected: it will eventually become concrete.

Given the notion of types with concrete members – which was not necessary at the lower level since terms may not contain ab-

[1] This is a slight simplification of νObj, where S_1 need not be a subtype of S_2. νObj's composition operator requires the self type for the composition to be specified explicitly. This new self type must be a subtype of the S_i.

$\boxed{\Gamma \vdash T <: T'}$ T is a subtype of T'

$$\frac{\Gamma \vdash S <: T \quad \Gamma \vdash T : K \quad \Gamma \vdash T <: T'}{\Gamma \vdash S <: T'} \text{ ST\_TRANS}$$

$$\frac{\Gamma \vdash S : K \quad \Gamma \vdash S \prec\!\!< R}{\Gamma \vdash S <: R} \text{ ST\_EXP}$$

$$\frac{\Gamma \vdash S : K \quad \Gamma \vdash T : K \quad \Gamma \vdash S \cong T}{\Gamma \vdash S <: T} \text{ ST\_EQ}$$

$$\frac{\Gamma \vdash T \ni \mathbf{type}\, L : K \setminus\!\setminus x \quad \Gamma \vdash K <: \mathbf{In}(\_, S)}{\Gamma \vdash T \# L <: S} \text{ ST\_ABS\_UPPER}$$

$$\frac{\Gamma \vdash T \ni \mathbf{type}\, L : K \setminus\!\setminus x \quad \Gamma \vdash K <: \mathbf{In}(S, \_)}{\Gamma \vdash S <: T \# L} \text{ ST\_ABS\_LOWER}$$

$$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash T_1 \lhd \{\mathbf{type}\, L = U\} <: T_2 \lhd \{\mathbf{type}\, L = U\}} \text{ ST\_INVAR}$$

$$\frac{\Gamma \vdash S <: S_2 \quad \forall j \in 1..k.\, \exists i \in 1..n.\, (m_i \overset{\text{label}}{\equiv} m'_j \wedge \Gamma \vdash m_i <: m'_j) \quad \forall i \in 1..n.\, (m_i\, \mathbf{deferred} \Rightarrow \exists j \in 1..k.\, m_i \overset{\text{label}}{\equiv} m'_j)}{\Gamma \vdash \{x : S \Rightarrow m_1 .. m_n\} <: \{x : S_2 \Rightarrow m'_1 .. m'_k\}} \text{ ST\_R}$$

$$\frac{\Gamma \vdash T_1 \prec\!\!< \{x : \_ \Rightarrow m_1 .. m_n\} \quad \Gamma \vdash T_2 \prec\!\!< \{x : \_ \Rightarrow m'_1 .. m'_k\} \quad \forall i \in 1..k.\, (m'_i\, \mathbf{deferred} \Rightarrow \exists j \in 1..n.\, m'_i \overset{\text{label}}{\equiv} m_j)}{\Gamma \vdash T_1 \,\&\, T_2 <: T_1} \text{ ST\_IELIMR}$$

$$\frac{\Gamma \vdash T_1 \prec\!\!< \{x : \_ \Rightarrow m_1 .. m_n\} \quad \Gamma \vdash T_2 \prec\!\!< \{x : \_ \Rightarrow m'_1 .. m'_k\} \quad \forall i \in 1..n.\, (m_i\, \mathbf{deferred} \Rightarrow \exists j \in 1..k.\, m_i \overset{\text{label}}{\equiv} m'_j)}{\Gamma \vdash T_1 \,\&\, T_2 <: T_2} \text{ ST\_IELIML}$$

$$\frac{\Gamma \vdash T <: T_1 \quad \Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \,\&\, T_2} \text{ ST\_IINTRO}$$

$$\frac{\Gamma \vdash T : K \quad T \text{ not an un-type}}{\Gamma \vdash T <: \mathbf{Any}} \text{ ST\_ANY}$$

$$\frac{\Gamma \vdash T : K}{\Gamma \vdash \mathbf{Nothing} <: T} \text{ ST\_NOTHING}$$

$$\frac{\Gamma \vdash T <: S}{\Gamma \vdash \mathbf{Un}[S] <: \mathbf{Un}[T]} \text{ ST\_UN}$$

Figure 8. Subtyping

$\boxed{\Gamma \vdash T : K}$ T has kind K

$$\frac{R \equiv \{x : S \Rightarrow m_1 .. m_n\} \quad \forall i \in 1..n.\, \Gamma, x : S \vdash m_i\, \mathbf{WF} \quad m_1 .. m_n\, \mathbf{noDuplicates}}{\Gamma \vdash R : \mathbf{Struct}\,(R)}\ \text{K\_R}$$

$$\frac{\Gamma \vdash \{x : \{x \Rightarrow m_1 .. m_n\} \Rightarrow m_1 .. m_n\} : K}{\Gamma \vdash \{x \Rightarrow m_1 .. m_n\} : K}\ \text{K\_RX}$$

$$\frac{\Gamma \vdash T_1 : \mathbf{Struct}\,(\{x : S_1 \Rightarrow \overline{m_i}^{\,i}\}) \quad \Gamma \vdash T_2 : \mathbf{Struct}\,(\{x : S_2 \Rightarrow \overline{m'_j}^{\,j}\}) \quad \forall i \in 1..n.\, \forall j \in 1..k.\, (m_i \overset{\text{label}}{=} m'_j \Rightarrow \Gamma \vdash m'_j <: m_i) \quad \overline{m''_k}^{\,k} = \overline{m_i}^{\,i} \uplus \overline{m'_j}^{\,j} \quad \Gamma \vdash S_2 <: S_1}{\Gamma \vdash T_1 \,\&\, T_2 : \mathbf{Struct}\,(\{x : S_2 \Rightarrow \overline{m''_k}^{\,k}\})}\ \text{K\_MIX}$$

$$\frac{\Gamma \vdash T : \mathbf{Struct}\,(\{x : S \Rightarrow m_1 .. m_n\}) \quad \Gamma \vdash \square\, T <: S \quad \forall i \in 1..n.\, m_i\, \mathbf{nonAbstract}}{\Gamma \vdash T : \mathbf{Concrete}\,(\{x : S \Rightarrow m_1 .. m_n\})}\ \text{K\_CONCRETE}$$

$$\frac{\Gamma \vdash p : T \quad \Gamma \vdash T : \mathbf{Struct}\,(\{x : S \Rightarrow m_1 .. m_n\})}{\Gamma \vdash p\,.\,\mathbf{type} : \mathbf{Concrete}\,(\{x : S \Rightarrow m_1 .. m_n\})}\ \text{K\_SING}$$

$$\frac{\Gamma \vdash p\,.\,\mathbf{type} : \mathbf{Concrete}\,(\{x \Rightarrow \mathbf{type}\, L : K\})}{\Gamma \vdash p\,.\,\mathbf{type} \# L : [\, x \mapsto p\,]\, K}\ \text{K\_SELPATH}$$

$$\frac{\Gamma \vdash T : \mathbf{Concrete}\,(\{x \Rightarrow \mathbf{type}\, L : K\}) \quad x \notin \mathbf{FV}(K)}{\Gamma \vdash T \# L : K}\ \text{K\_SEL}$$

$$\frac{\Gamma \vdash T : \mathbf{Struct}\,(\{x : S \Rightarrow m_1 .. m_n\}) \quad \exists i \in 1..n.\, m' = \mathbf{refines}(m_i, cm) \quad \Gamma, x : S \vdash m'\, \mathbf{WF} \quad m''_1 .. m''_n = m_1 .. m_n \uplus m' \quad x \notin \mathbf{FV}(cm)}{\Gamma \vdash T \lhd \{cm\} : \mathbf{Struct}\,(\{x : S \Rightarrow m''_1 .. m''_n\})}\ \text{K\_RFN}$$

$$\frac{\Gamma \vdash T : K_1 \quad \Gamma \vdash K_1 <: K_2}{\Gamma \vdash T : K_2}\ \text{K\_SUBSUME}$$

$$\frac{}{\Gamma \vdash \mathbf{Any} : \mathbf{Struct}\,(\{x \Rightarrow\})}\ \text{K\_ANY}$$

$$\frac{\Gamma \vdash R : \mathbf{Struct}\,(R)}{\Gamma \vdash \mathbf{Nothing} : \mathbf{Struct}\,(R)}\ \text{K\_NOTHING}$$

$$\frac{\Gamma \vdash T : K}{\Gamma \vdash \mathbf{Un}\,[\, T\,] : \mathbf{Struct}\,(\{x \Rightarrow\})}\ \text{K\_UN}$$

Figure 9. Classifying Types

$\boxed{\Gamma \vdash m <: m'}$ m is a submember of m'

$$\frac{\Gamma \vdash T <: T'}{\Gamma \vdash \mathbf{val}\, l : T\_ <: \mathbf{val}\, l : T'\_}\ \text{SM\_VAL}$$

$$\frac{\Gamma \vdash K <: K'}{\Gamma \vdash \mathbf{type}\, L : K\_ <: \mathbf{type}\, L : K'}\ \text{SM\_TYPEA}$$

$$\frac{}{\Gamma \vdash \mathbf{type}\, L : K = \_ <: \mathbf{type}\, L : K = \_}\ \text{SM\_TYPEC}$$

Figure 10. Subtyping for members

stract members – type member selection is classified analogously to value member selection. Type refinement is almost literally the same as at the term level, as is subsumption.

Finally, the top and the bottom of the subtype lattice must be classified, as well as un-types. **`Any`**, and certainly **`Nothing`**, are not essential to the type system. However, **`Any`** is needed to be able to select a member on the universe (the top-level object): that member must have a type that does not depend on the universe's self variable, but all user-defined types are (indirectly) selected on the universe's self type. Since **`Any`** exists outside of the user-defined universe, it can serve this purpose. An alternative would be to introduce another variable binding construct, such as `let`. **`Nothing`** is used as the default lower bound of the interval kind. It may be given the same kind as any well-kinded structural type.

4.4 Subkinding

Subkinding (Fig. 12) introduces contravariance for un-kinds (SK_UN), so that type un-members conform contravariantly. Other than that, the relation defines a simple lattice, with the interval kind at the top.

A nominal type can be subsumed to a structural one (SK_NOM) – but not vice versa! A concrete type is also a structural one (SK_CONC). A structural type that includes all the members in `R`, is thus in the interval (**`Nothing`**, `R`) (SK_STRUCT).

Interval inclusion gives rise to subkinding (SK_CTX_IN). The kinds that classify structural types and concrete types have similar subsumption properties based on subtyping (SK_CTX_CONC, SK_CTX_STRUCT).

5. Design Space

After introducing Scalina in detail, we look at the bigger picture by briefly positioning its abstraction mechanisms in the design space. Scalina's main goal is to provide the essential features to model an object-oriented language – such as objects with named members, mutual recursion through the self variable, and mixin composition – while also allowing functional concepts to be encoded with the same safety guarantees as in functional calculi.

For an expedient exploration of the design space of abstraction mechanisms, we shall restrict ourselves to investigating the instantiations of the following question: "Is a *term/type* that abstracts from a *term/type* using *a parameter/an abstract member* a first-class *term/type*?" We will answer these questions for Java, Scala, System F^{sub}_{ω}, and Scalina.

Note that we use 'term' to denote anything that resides at the 'base' level, such as an object in OO, or a function in FP. We do not imply any connection to syntactic terms. A 'type' is something that classifies terms, and thus resides at the next level. We use 'entity' to mean either a term or a type, when it only matters that the denoted entity can perform computation. Finally, a 'classifier' classifies an entity: a type classifies a term, and a kind classifies a type.

Table 1 gives an overview of the analysis discussed below. The row of an entry determines what is abstracted from, and the column denotes the level of the abstraction. When the constructs in a part of the table are not all first-class constructs, (+) and (-) are used to make the distinction. We consider a construct "first-class" if it can be abstracted over. '/' means 'not supported'. The superscripts in parentheses are intended to aid the reader in correlating the schematic representation in table 1 and the following discussion.

In Java, a term may only abstract from a term[1] or a type[2] using parameterisation (functional abstraction): as already mentioned, a method abstracts from the concrete values of its arguments, but a method is not a first-class term in Java. Similarly, a polymorphic method may have type parameters, but again, such a method is not a first-class entity. Terms cannot have abstract members[3, 4].

At the type level, still in Java, a constructor argument[5] can be considered as parameterising a type in a value (a constructor, like a method, is not a first-class term). Since Java 5.0, a class (a type) may take type parameters[6], but a parameterised type is not a first-class type unless it is fully applied. Finally, a class with an abstract method[7] is a first-class type that abstracts from a term. When deciding whether a type is first-class, we do not take into account whether it may be instantiated.

Scala introduces several improvements over Java. Firstly, λ-abstraction[8] is directly supported, thus a term abstracting from a term is a value. Secondly, we recently implemented direct support for type constructor polymorphism in Scala 2.5, so that a parameterised type[9] is considered a first-class (higher-kinded) type [19]. Finally, a class may have abstract *type* members[10].

System F_{ω}^{sub} is a purely functional calculus. Naturally, we only consider its support for abstraction using parameterisation. A term that abstracts from a term is written as $\lambda x : T.t$[11]. A term may also be parametric in a type: $\lambda X : K.t$[12]. A type abstracts from a type as $\lambda X : K.T$[13]. To abstract from terms at the level of types[14], we must turn to dependently typed versions of the calculus.

The overview in table 1 contains a striking void in the quadrant of terms with abstract members[3,4,15,16].

Nevertheless, Self, one of the earliest OO languages, represents a method as an object with "argument slots" [30]. In other words, a method is a first-class term (i.e., an object) that uses abstract members to abstract from other terms.

Finally, Scalina's object-oriented abstraction mechanisms are split out with respect to the clients they cater to. An object can abstract over an object that is to be supplied by an external client using a value un-member[17]. A type may abstract over a value in the same way [18]. Since objects are not allowed to have abstract members, they cannot abstract over terms[19] or types[20] for internal clients. Types, on the other hand, may contain abstract value[21] or type[22] members. Finally, an object[23] or a type[24] type can abstract over a type using a type un-member.

Thus, our brief survey has shown that Scalina supports all variations of abstraction mechanisms that are used in practice, without admitting too many features that do not appear in a full language. We designed Scalina so that it includes the main concepts of object-oriented languages, such as objects with named members and mutual recursion through a self variable, mixin composition, subtyping, and lightweight dependent types. Furthermore, although Scalina does not contain any mechanisms for parameterisation, it can safely and straightforwardly encode functional-style abstraction as well. Others have studied the advantages of OO-style abstraction over the functional style, and vice versa [6, 29, 12, 13].

Construct	...in term (1st class?)	...in type (1st class?)
Java		
Parameter (FP)		
Term	method (-) [1]	constructor (-) [5]
Type	method (-) [2]	generic class (-) [6]
Abs. mem. (OO)		
Term	/ [3]	class w/abs. method (+) [7]
Type	/ [4]	/
Scala		
Parameter (FP)		
Term	method (-) anon. function (+) [8]	constructor (-)
Type	method (-)	generic class (+) [9]
Abs. mem. (OO)		
Term	/ [15]	abs. **`val`**/`def` (+)
Type	/ [16]	abs. type member (+) [10]
System F_{ω}^{sub}		
Term	$\lambda x : T.t$ [11]	/[14]
Type	$\lambda X <: T.t$ [12]	$\lambda X :: K.T$ [13]
Scalina		
Term (ext.)	obj. w/value un-member [17]	type w/value un-member [18]
Term (int.)	/ [19]	type w/value abs. mem. [21]
Type (ext.)	obj. w/type un-member [23]	type w/type un-member [24]
Type (int.)	/ [20]	type w/type abs. mem. [22]

Table 1. Abstraction mechanisms: overview
(The superscripts link the entries in the table to the relevant part of the discussion.)

6. Encoding System F_{ω}^{sub}

Table 2 shows how terms, types and kinds from System F_{ω}^{sub} [22, Ch. 31] can be encoded in Scalina. Using Pierce's terminology, an abstraction is modelled as an object with an un-member `a` that represents the argument, and a member `apply` that encodes the body of the abstraction. Note that we have to infer the type `T'`. Application is decomposed into refining the `a` un-member with the encoding of the actual argument, and selecting the `apply` member.

The encoding of a polymorphic value re-uses the pattern we used for term abstraction, except that the argument is now a type un-member instead of a term-level one. We use an interval kind to model type bounds: '$<: T$' becomes '`: In(Nothing, [[T]])`'. Type application does not present new challenges.

At the level of types, function types and universal types become the obvious structural types, which we established when encoding (polymorphic) function values. Similarly, we simply hoist our term-level abstraction and application to the type level to replace operator abstraction and application. The kind-level is easily derived from the type that encodes operator abstraction.

The evaluation of the encoding of a value application proceeds by E_RFN pushing the refinement of the object to the object's type, so that E_SEL can look up the `apply` member in the type that now has a concrete value for it. Evaluating a type application also uses E_RFN to push the concrete type information into the type of the value, which tracks the value's members. However, this binding is never used during later evaluation steps, as the only types that interact with evaluation, are those that can be used to instantiate a new object. These types must statically expand to a structural type, which is not possible for type un-members.

Finally, we note that the contravariant rule for un-member conformance means that Scalina can encode full System F_{ω}^{sub}, and that the undecidability of the latter should thus carry over to Scalina. We defer a more formal account of the correspondence with System F_{ω}^{sub} to future work.

7. Meta-theory

The traditional term-level safety proofs show that it suffices to type check a program once in order to guarantee certain properties for every possible evaluation trace. In Scalina, we ensure that member lookup never fails to find the required label with the corresponding right-hand side, and that an un-member is at most refined once on the same object.

The type-level guarantees are similar, though more subtle. Since type selection is only well-kinded if the target of the selection is known to become a concrete type during type checking, we ensure that selection can always proceed on types of kind $\mathbf{Concrete}(R)$. Note that we consider certain other type selections, such as selecting a nominal type, to be in canonical form, so that this kind of selection is not expected to proceed.

We are actively working on the proofs of the meta-theory and their precise formulation.

$[\![t]\!]^{t'}$	$\equiv$	replace the free variable in the encoding of t with t'
$[\![\lambda x : T.t]\!]$	$\equiv$	`new {self ⇒ val a: Un[[[T]]]; val apply: T' = [[t]]^self.a}`
$[\![t\, t']\!]$	$\equiv$	`([[t]] ◁{val a = [[t']]}).apply`
$[\![\lambda X <: T.t]\!]$	$\equiv$	`new {self ⇒ type a: Un[In(Nothing, [[T]])]; val apply: T' = [[t]]^self.a}`
$[\![t\,[T]]\!]$	$\equiv$	`([[t]] ◁{type a = [[T]]}).apply`
$[\![\text{Top}]\!]$	$\equiv$	`Any`
$[\![T \to T']\!]$	$\equiv$	`{val a: Un[[[T]]]; val apply: [[T']]}`
$[\![\forall X <: T.T']\!]$	$\equiv$	`{type a: Un[In(Nothing, [[T]])]; val apply: [[T']]^self.a}`
$[\![\lambda X :: K.T]\!]$	$\equiv$	`{type a: Un[[[K]]]; type apply : K' = [[T]]^self.a}`
$[\![T\, T']\!]$	$\equiv$	`([[T]] ◁{type a = [[T']]}).#apply`
$[\![*]\!]$	$\equiv$	`Struct({x ⇒ })`
$[\![K \Rightarrow K']\!]$	$\equiv$	`Struct({self ⇒ type a: [[K]]; type apply: [[K']]})`

Table 2. Informal encoding of System F^{sub}_{ω} syntax in Scalina

$\boxed{\Gamma \vdash m\ \mathbf{WF}}$ m is well-formed

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathbf{val}\, l : T = t\ \mathbf{WF}}\ \text{M\_VALC}$$

$$\frac{\Gamma \vdash T : \mathbf{Struct}\,(R)}{\Gamma \vdash \mathbf{type}\, L : \mathbf{Nominal}\,(R) = T\ \mathbf{WF}}\ \text{M\_TYPENOM}$$

$$\frac{\Gamma \vdash T : K}{\Gamma \vdash \mathbf{type}\, L : K = T\ \mathbf{WF}}\ \text{M\_TYPEC}$$

$$\frac{\Gamma \vdash T : K}{\Gamma \vdash \mathbf{val}\, l : T\ \mathbf{WF}}\ \text{M\_VALA}$$

$$\frac{\Gamma \vdash K\ \mathbf{WF}}{\Gamma \vdash \mathbf{type}\, L : K\ \mathbf{WF}}\ \text{M\_TYPEA}$$

Figure 11. Well-formedness of members

$\boxed{\Gamma \vdash K <: K'}$ K is a subkind of K'

$$\frac{\Gamma \vdash K_2 <: K_1}{\Gamma \vdash \mathbf{Un}\,[\,K_1\,] <: \mathbf{Un}\,[\,K_2\,]}\ \text{SK\_UN}$$

$$\frac{}{\Gamma \vdash \mathbf{Nominal}\,(R) <: \mathbf{Struct}\,(R)}\ \text{SK\_NOM}$$

$$\frac{}{\Gamma \vdash \mathbf{Concrete}\,(R) <: \mathbf{Struct}\,(R)}\ \text{SK\_CONC}$$

$$\frac{}{\Gamma \vdash \mathbf{Struct}\,(R) <: \mathbf{In}\,(\mathbf{Nothing}\,,\ R)}\ \text{SK\_STRUCT}$$

$$\frac{\Gamma \vdash R_1 <: R_2}{\Gamma \vdash \mathbf{Concrete}\,(R_1) <: \mathbf{Concrete}\,(R_2)}\ \text{SK\_CTX\_CONC}$$

$$\frac{\Gamma \vdash R_1 <: R_2}{\Gamma \vdash \mathbf{Struct}\,(R_1) <: \mathbf{Struct}\,(R_2)}\ \text{SK\_CTX\_STRUCT}$$

$$\frac{\Gamma \vdash T_2 <: S_2 \quad \Gamma \vdash S_1 <: T_1}{\Gamma \vdash \mathbf{In}\,(T_1\,,\ T_2) <: \mathbf{In}\,(S_1\,,\ S_2)}\ \text{SK\_CTX\_IN}$$

Figure 12. Subkinding

8. Related Work

8.1 Safe type-level abstraction

Since the seminal work of Girard and Reynolds in the early 1970's, fragments of the higher-order polymorphic lambda calculus or System F_ω [15, 24, 5] have served as the basis for many programming languages. Furthermore, the interaction between higher-kinded types (types with un-members) and subtyping is a well-studied subject [23, 9]. A similarity of interest is Cardelli's notion of power type [8], which corresponds to Scalina's `In(S, T)` kind.

Despite the vast volume of work on type-level abstraction in functional programming languages, object-oriented languages offer comparatively limited support. Most OO languages do provide parametric polymorphism [4], but few give type constructors first-class status. Cremet and Altherr extend Featherweight Generic Java with higher-kinded types [3]. To the best of our knowledge, besides OCaml [17, 6.8.1], Scala is the only OO language with support for type constructor parameters [19]. Of course, this can be encoded in languages with abstract type members, such as gbeta [11].

We briefly mention type-level computation [27, 28, 25], which can be used to enforce properties of term-level programs. The traditional term-level programmer need not be the one who designed the type-level machinery to enforce these properties.

8.2 Modelling OO

Given the wealth of research on extensions of the λ calculus, it is only natural that studies of the essence of object-oriented languages build on these ideas. Even though encoding objects requires a lot of extra machinery, such as records, subtyping, and recursive types, this complexity is probably inherent. However, modelling OO using a combination of FP *and* OO seems to fail Occam's razor. Nonetheless, a lot of object calculi fall in this category [16, 14, 10].

The other side of the spectrum – using a purely object-oriented calculus without FP concepts – can be traced back to Abadi and Cardelli's seminal work [1, 2]. However, in their first-order system, "an object type is invariant in its component types". Thus, object types cannot encode function types in the presence of subtyping, as the latter require a mix of contravariance and covariance. To solve this, they introduce universal and existential quantification in their second-order system. Universal quantification, like un-members, behaves contravariantly. Similarly, existential quantification introduces covariance, which we allow for normal members.

In other respects, Abadi and Cardelli's first-order system is more powerful than Scalina: our refinement operator does not allow recursion through the self variable. However, this limitation simpli-

fies the calculus without ruling out refinement's primary use, which is similar to function application: supplying a value to a function does not rely on the values supplied earlier.

To further the similarity with application, refinements do not require type or kind annotations for the supplied entities. Nonetheless, it is possible to have type un-members that classify value un-members: the type un-members must then be refined before the value un-members, because the supplied types determine the acceptable values for the value un-members. Note that we use first-class types, which do have a self variable, for overriding.

Scalina was directly inspired by the νObj calculus [21]. The main difference is that Scalina introduces un-members and refinement at the term and type level. νObj uses class templates for term-level abstraction, and only provides covariant abstract type members for type-level abstraction. The latter implies that well-formed type-level applications may surprise the type function with unexpected arguments. It is important to note that this does not have any impact on the run-time behaviour of such programs. It does however fail to provide the type-level equivalent of the guarantees that term-level abstraction builders have come to rely on.

9. Conclusion and Ongoing Work

The immediate goal of Scalina is to provide a foundation for proving our extension of Scala with type constructor polymorphism sound. In this paper, we have shown how our calculus improves over the νObj calculus with respect to safe type-level abstraction. More specifically, we formulated the notion of *kind soundness*, which ensures safety for type-level abstractions. To achieve this, we distinguish "input" and "output" members. Given the covariant nature of Scala's abstract type members, they should only be used for output. We introduce un-members to deal with input. Furthermore, we illustrated Scalina's uniform, and purely object-oriented, treatment of term-level and type-level abstractions as first-class entities.

Although we are well on our way to proving Scalina sound at both levels, the meta-theory is not yet complete. Once these results have been established, we will define a type-preserving translation from an essential subset of Scala with type constructor polymorphism into Scalina. Similarly, the full correspondence with variants of System F_{ω}^{sub} remains an interesting topic to explore.

A broader perspective on our work is that more powerful type-level abstractions are an important tool in improving the robustness of software written in tomorrow's languages. In order to make it practical to use these techniques, we must be able to write type-level abstractions once and re-use them safely in different settings.

Acknowledgments

The authors would like to thank Erik Ernst for his extremely thorough feedback on earlier drafts of this paper. Furthermore, we gratefully acknowledge Dave Clarke, Burak Emir, Bart Jacobs, Sean McDirmid, and Marko van Dooren for their insightful comments and interesting discussions. Finally, we thank the anonymous reviewers for their helpful suggestions and insightful comments. We typeset Scalina's theory using OTT (with `ottlayout.sty`) [26].

References

[1] M. Abadi and L. Cardelli. A theory of primitive objects: Second-order systems. *Sci. Comput. Program.*, 25(2-3):81–116, 1995.

[2] M. Abadi and L. Cardelli. A theory of primitive objects: Untyped and first-order systems. *Inf. Comput.*, 125(2):78–102, 1996.

[3] P. Altherr and V. Cremet. Adding type constructor parameterization to Java. Accepted to the workshop on Formal Techniques for Java-like Programs (FTfJP'07) at the European Conference on Object-Oriented Programming (ECOOP), 2007.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, pages 183–200, 1998.

[5] K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Inf. Comput.*, 85(1):76–134, 1990.

[6] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In E. Jul, editor, *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549. Springer, 1998.

[7] P. S. Canning, W. R. Cook, W. L. Hill, W. G. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, pages 273–280, 1989.

[8] L. Cardelli. Structural subtyping and the notion of power type. In *POPL*, pages 70–79, 1988.

[9] A. B. Compagnoni and H. Goguen. Typed operational semantics for higher-order subtyping. *Inf. Comput.*, 184(2):242–297, 2003.

[10] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for Scala type checking. In R. Kralovic and P. Urzyczyn, editors, *MFCS*, volume 4162 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2006.

[11] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[12] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, 2001.

[13] E. Ernst. Reconciling virtual classes with genericity. In D. E. Lightfoot and C. A. Szyperski, editors, *JMLC*, volume 4228 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2006.

[14] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL*, pages 171–183, 1998.

[15] J. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These d'Etat, Paris VII, 1972.

[16] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[17] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system*, 2007. release 3.10.

[18] E. Meijer. Confessions of a used programming language salesman getting the masses hooked on Haskell. In *OOPSLA 2007: Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2007. ACM Press.

[19] A. Moors, F. Piessens, and M. Odersky. Towards equal rights for higher-kinded types. Accepted for the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages at the European Conference on Object-Oriented Programming (ECOOP), 2007.

[20] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[21] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In L. Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2003.

[22] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[23] B. C. Pierce and M. Steffen. Higher-order subtyping. *Theor. Comput. Sci.*, 176(1-2):235–282, 1997.

[24] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[25] T. Schrijvers, M. Sulzmann, S. Peyton-Jones, and M. Chakravarty. Towards open type functions for Haskell. In *Proc. of IFL 2007*, 2007.

[26] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. S. sa. Ott: Effective tool support for the working semanticist. In *Proceedings of ICFP 2007: the 12th ACM SIGPLAN International Conference on Functional Programming (Freiburg)*, Oct. 2007. 12pp.

[27] T. Sheard. Type-level computation using narrowing in Ωmega. In *PLPV*, 2006.

[28] M. Sulzmann, J. Wazny, and P. J. Stuckey. A framework for extended algebraic data types. In M. Hagiya and P. Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 47–64. Springer, 2006.

[29] K. K. Thorup and M. Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In R. Guerraoui, editor, *ECOOP*, volume 1628 of *Lecture Notes in Computer Science*, pages 186–204. Springer, 1999.

[30] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA*, pages 227–242, 1987.

[31] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.