Nested Types

Martin Odersky and Christoph Zenger
Swiss Federal Institute of Technology Lausanne

Preliminary Version, January 16, 2001

1 Introduction

We currently see a convergence of module systems and type systems for object-oriented programming. This convergence is driven by the demands of large scale component systems. Object-oriented languages for medium-scale systems such as Eiffel or Sather still worked with an unstructured set of classes, but as systems grow this becomes less practical. Consequently, more recent languages such as Beta [MMPN93] or Java [GJS96] allow for class nesting. Class nesting raises issues such as hiding and refinement that have first been studied in module systems. On the other hand, recent module systems embody many of the properties of class systems, e.g. first-class modules[Rus98], mixin modules [Dug96], or recursion between modules.

This development is pushing the limits of what is expressible in current type systems. One possible approach is to combine known type constructs for modules and objects into a language which has both. Examples of such amalgamations are Moby[FR99] and OCaml[Ler99]. While workable, this approach tends to lead to a duplication of aspects inherent in modules and objects which often overlap. Another approach with better synergy potential is to unify the object and module worlds by identifying corresponding concepts. The most natural such identification, proposed by Cardelli[Car89] identifies modules with objects, functors with classes, and signatures with object types. However, a realization of this proposal in statically typed programming languages has remained incomplete to-date. We believe the reason for this stall is that the traditional type systematic foundations of modules and objects have been largely incompatible.

The module world concentrates on abstraction by type hiding and propagation of partial type information. For a long time, the object world made to with much cruder hiding constructs such as the private, public and protected access specifiers in C++ or Java. On the other hand, object-oriented languages naturally assume powerful forms of composition, such as general recursion between classes, inheritance, and first-class functions over objects. So far it has not been easy to extend the type-theoretic foundations of module languages to these forms of composition.

In this paper we propose the concept of nested types as a basis for a unification of the object and module worlds. Nested types support the abstraction properties of ML-style module systems without requiring the heavy-theoretic machinery of dependent function types. At the same time, nested types naturally support even advanced object-oriented composition constructions through simple encodings. We show in the paper how nested types can model module concepts such as abstract types, functors, and sharing constraints. We also show how they can model object-oriented concepts such as inheritance through delegation, abstract methods, self types with matching, and virtual types [MMP89, Tor98, IP99, BOW98]. These encodings are all presented using the – as yet experimental – programming language Funnel [Ode00a, AOS+00].

Nested types are formalized in a kernel type system. Deviating from long-standing practice in type theory research, the type system presented here is *nominal*. That is, record types need to be introduced by explicit declaration. A declared record type is incompatible with other unrelated declared record types even if the two types have the same structure. Furthermore, a record type is a direct subtype only of those types given in an explicit supertype clause of its definition, with subtyping being the transitive closure of this relation. While this represents a departure from usual tehoretical approaches in type systems, it is very close to the object-oriented programming languages we want to model. Unlike other previous approaches to model object-oriented languages such as FJ [IPW99] or other lightweight versions of Java [OW97, DE97, NvO98, FKF98], our type system does allow local type declarations and models well-formedness of type declarations using inference rules, rather than assuming a global well-formed class graph.

A debate about the relative merits of structural and nominal type systems, while interesting, is not the purpose of this paper. We chose a nominal type system for Funnel mainly because it enables certain notions of extensibility that we are interested in [Ode00b]. Another important advantage of nominal type systems is that the explicit type definitions sidestep the undecidability problems which otherwise arise for type systems with bounded quantification [Pie92]. We conjecture our type system to be decidable, although an algorithm and a formal proof of its correctness and completeness are still missing at the present time. A third advantage is that nominal type systems make it natural to consider recursion between types from the start instead of introducing it as an afterthought. In structural type systems, recursion seems to interacts poorly with many of the more advanced type construction needed to deal with objects [IP99].

The rest of this paper is structured as follows. Section 2 motivates nested types by reviewing existing type-theoretic approaches to ML-style module systems. Section 3 explains informally the key aspects of nested types. Section 4 discusses encodings of object-oriented constructs into our base language. Section 5 concludes. The appendix describes and explains our nominal type system for nested types.

2 Types for Modules

Given the amount of solid work that has gone into ML-style module systems, one would expect that their concepts would have been picked up readily by the object oriented community. In this section we review the existing most popular type theoretic models of module systems, and discuss possible impediments for their adoption in object-oriented programming languages. The review of the state of the art then motivates the introduction of nested types. Even though this section is largely about SML[AM91, MTHM97], we will explain all examples using Funnel syntax to stay consistent with the exposition in the rest of the paper.

To understand the different approaches to modular abstraction, consider a type (or, in ML parlance, a signature) S, which contains an abstract type t and fields inc and zero which have types referring to t.

type S = {	**module** M: S = {
type t	**type** t = Int
val inc: t → t	**val** inc (x: t) = x + 1
val zero: t	**val** zero = 0
}	}

The module definition of M defines a value which implements the abstract signature. In their seminal paper [MP85], Mitchell and Plotkin have shown that abstract types correspond to existential types in second-order lambda calculus. In their interpretation, the type S would correspond to the existential type $\exists$t.{inc: t → t, zero: t}. However, the existential types interpretation of abstract types has not been adopted in SML, because it destroys too much information. For instance, the expression M.inc(M.zero) would not be typable in the classical existential interpretation since the two occurrences of M would have types with different existentially quantified variables. Cardelli and Leroy have proposed with the dot notation[CL90] a way to recover the missing information. Their scheme opens the scope of an existential as soon as a value is defined. Thus, the type M.t would be assigned to M.inc(M.zero). The qualified M.t acts as a skolem constant that names the abstract component t of M.

Things become more complicated if we introduce functions over records (functors in SML). Let's try to write a function takeZero that retrieves the zero field of values of type S. The question is, which type should we give to that function?

def takeZero (x: S): ? = x.zero

SML's answer would be: **def** takeZero (x: S): x.t. That is takeZero would be given the *dependent function type* $\Pi(x : t)(x.t)$. This propagates all the necessary information about the functor argument type into the functor result type. In particular, we have takeZero(M): M.t. Hence, an expression such as M.f(takeZero(M)) would be well-formed.

Dependent function types alone are not sufficient to accurately describe all implementation sharing. As an example, consider function incZero which applies the inc function of one implementation of S to the zero field of another.

def incZero (x: S, y: S) = x.inc(y.zero)

The above function is not typable in SML, because the types x.t and y.t are unrelated. This is required to maintain type soundness. If incZero was well-formed, it could be used at run-time to mix two different incompatible S-implementations for x and y.

The last example is an instance of the "diamond import" problem [Mac86], which suggested the introduction of sharing constraints. With the sharing constraints of original SML [AM91], incZero could be typed as follows.

def incZero (x: S, y: S **sharing** x.t = y.t) = x.inc(y.zero)

The unification-based sharing constraints of original SML have been superseded in SML 97 [MTHM97] by translucent sums [HL94] and – closely related – manifest types [Ler94], which give a more logical approach to sharing and also have better properties for separate compilation. Using manifest types, the incZero example would be formulated as follows.

def incZero (x: S, y: S { **type** t = x.t }) = x.inc(y.zero)

In other words, we constrain the y parameter to range over subtypes of S which have a type component t bound to x.t.

Dependent function types with translucent sums fit the bill for an adequate description of information propagation whrough functors. But their heavy type-theoretic machinery makes their adaptation in a system with unified objects and modules a daunting task. Maybe because of this complexity, no completely unified proposal has emerged to date.

An alternative approach is to research type-theoretic foundations of modules that are simpler than dependent types. One such foundation has been given by Claudio Russo in his thesis[Rus98], where he describes the static semantics of modules using only existential and universal types. In his proposal, takeZero would be represented with the following internal signature:

def takeZero [t'] (x: {**type** t = t', **val** inc: t → t, **val** zero: t}): t' = m.zero

That is, we "lift" the existential quantifier of S out to the toplevel where it becomes a universal quantifier t' of the type of the enclosing function, takeZero.[1] Keeping the dot notation as a syntactic convenience, takeZero can then be applied to the type argument [M.t] and the value argument [M]:

[1] [...] brackets denote type parameterization in Funnel.

```
def takeZero [M.t] (M)   : M.t
```

The lifting technique has no problems with representing sharing constraints. For instance, function incZero can be represented as follows.

```
def incZero [t'] (x: {type t = t', val inc: t → t, val zero: t},
                  y: {type t = t', val inc: t → t, val zero: t}): t'  =  x.inc (y.zero)
```

One shortcoming of Russo's technique is that it involves the manipulation of possibly large structures of types. When calling a function with structure S, every existentially bound type variable in S and its sub-signatures has to be lifted out. The lifting operations are in conflict with the role of signatures as packaging constructs for types as well as values. Probably for this reason Russo does not propose his representation as an alternative to SML source syntax. He investigates instead mappings of SML source into his framework, so that his representation needs to be used only internally. This choice is not the only possible one; for instance Mark Jones [Jon96] proposes similar techniques as a way to express modular structure in the source language itself. But in our opinion the lifting techniques are notationally too heavy in programs where objects represented as modules are ubiquitous.

3 Principles of Nested Types

Is there a middle road between the type-theoretic complexities of dependent function types and the notational complexities of parametric polymorphism? This paper proposes nested types as a candidate type discipline and studies its application to typical object-oriented constructions. The idea is to introduce types as components of types, replacing the "types as components of values" view of dependent types and the dot notation.

We now introduce the principles of nested types and show how they can express the module concepts introduced in the last section. The discussion in this section is largely explanatory and informal; a formal treatment of nested types is deferred to the the appendix.

Our language is an applied variant of a higher-order mostly-functional language with a polymorphic type system with F-bounded quantification[CCH+89]. As a first approximation, it can be understood as a variant of system $F^{\omega}_{\leq}$, extended with (nominal) records and mutable variables. We often elide redundant type information, assuming that a local type inference system [PT98, OZZ01] is available to supply the omitted annotations.

A record type definition introduces a new record with given supertypes, type components and value components. Here is an example:

```
type R = S {
   type t = Int
   val f: t → t
   val z: t
}
```

This definition introduces the type R as a subtype of type S with type field t and value fields f and z. The type field t is defined as an alias of the type Int. Given this definition, the qualification R^t denotes the t component of type R. It is also possible to introduce new record types as components of types, as in:

```
type O = T, U { type I = { val x: t }}
```

Finally, it is possible to define an abstract type, as in:

```
type S = {
    type t
    val f: t → t
    val z: T
}
```

This introduces an abstract type field t, the implementation of which is hidden. In general an abstract type definition may be constrained by a supertype bound and/or nested definitions, as in:

```
type s <: S { type t = Int }
```

This declares s to be an abstract type of which it is known that it is a subtype of S and that it has a type field **type** t = Int.

When defining a subtype **type** R <: S { Ds }, we inherit all type and value fields of S in R unless a field of the same name if defined in R. If a field x: T in S is redefined as x:T' in Ds, we require that the redefined type is a subtype of the original type, i.e. T' <: T. Abstract type definitions **type** t <: T { Ds' } in S may also be redefined in Ds. The redefinition might define t to be another abstract type or a concrete type. In both cases the redefinition of t must make t a subtype of the bound T in the supertype which also defines all fields {Ds'} as given in S. For instance, the definition of R above is legal according to these rules. The following variant, however, exhibits a type error:

```
type AbsList {                    type BadList = AbsList {
    type t <: AbsList                 type t
    val next: () → t                  val next: () → t
}                                 }
```

The problem with this code is that the type BadList relaxes the constraints on the abstract t type defined in its supertype AbsList, rather than strengthening it, as would be required. It is also illegal to redefine a non-abstract type binding, except with the same type. For instance, the following definition would not be type-correct:

```
type RSub = R { type t = String }
```

The problem here is that the concrete type binding t = Int in the supertype R has been replaced with the binding t = String in the subtype RSub.

Definitions of record values require that the type of the record being defined is given explicitly. A definition of a record of type R contains a binding for every value field of R. Type fields, on the other hand, are not given, since they are conceptually components of types, not values. For instance, the following would define a value r of type R: [2]

```
val r: R = {
    def f(x) = x + 1
    val z = 0
}
```

Projection to a supertype is used for hiding, as for instance in: **val** s: S = r.

Sometimes it is tedious to define a separate type for every record, in particular for top-level records. We therefore introduce *modules* as syntactic sugar for record values without separately provided

[2] Funnel knows three kinds of value bindings. **val** x = E binds the name x to the result of evaluating E. Elaboration of the binding involves evaluation of E. **let** x = E is similar, except that E is not evaluated at the point of definition. It is instead evaluated lazily, once x is first dereferenced or applied. Finally **def** f(..) = E defines a (possibly parameterless) function, where E is evaluated every time f is applied.

types. A module simply introduces a record value with a new type. The record value is defined lazily. The main purpose of **module** over **let** is that the module's type does not have to be given explicitly, but is computed from the module's definitions. It follows that the definitions of a module need to be fully annotated with types. The name of the type of a module M is irrelevant; we use internally the name M$I for it. As an example, consider the module:

```
module M = {
  type T = Int
  def f (x: T): T = x + 1
  val z: T = 0
}
```

This would generate the following type M$I and record value M.

```
type M$I = {                    let M: M$I = {
  type T = Int                    def f (x) = x + 1
  def f (x: T): T                 val z = 0
  val z: T                      }
}
```

Subtyping and projection for modules are analogous to the definition of these concepts for records. For instance, one can write

```
module M: S = {
  type T = Int
  def f (x: T): T = x + 1
  val z: T = 0
}
```

as an abbreviation of

```
let M: AbsM = {
  module M = AbsM {
    type T = Int
    def f (x: T): T = x + 1
    val z: T = 0
  }
  M
}
```

The qualified type S^t conveys no information at all; it corresponds roughly to the completely abstract type $\exists t.t$. More useful types are obtained by introducing the concept of a *witness type*. If x is a term-identifier then the witness type x! represents the (generally unknown, but fixed) implementation type of x. Type components of witness types can be selected using (^), as usual. For instance, M!^t represents the (unknown) implementation of t in module M. Witness types are a generalization of the dot notation. Unlike the dot notation, witness types allow a dinstinction between the two conceptually separate concepts of naming an unknown implementation type and selecting a type component. For syntactic convenience, and to emphasize the relationship with the dot notation, we retain M.t as an abbreviation of M!^t.

With nested types, the expression M.inc(M.zero) type-checks as follows.

```
M                : M!
M.inc            : M!^t → M!^t
M.zero           : M!^t
M.inc(M.zero)    : M!^t
```

Thye witness type M! is essential in this example to establish the equality of the argument type of M.inc and the type of M.zero.

A witness type x! is valid only in the scope of x. Hence it would be illegal to write

```
def takeZero (x: S): x!^t = x.zero                    // illegal
```

since this would let escape the witness type x! out of the scope of parameter x into the result type of function takeZero (if such an escaping reference was legal we would in effect be back to dependent function types). However, we can write another version of the takeZero function as follows.

```
def takeZero [s <: S] (x: s): s^t = x.zero
```

This makes use of bounded quantification [s <: S] which lets s range over all subtypes of S. Similarly to the lifting technique of Russo we have expressed propagation of type identities by introducing a fresh type variable. But nlike the lifting technique our scheme only needs to quantify over subtypes of the parameter signature itself. There is no need to reach inside that signature in order to lift out quantifiers embedded in it. Hence, we avoid the potential blow-up in the size of types which is inherent in the lifting technique.

Nested types deal naturally with sharing. For instance, consider again the incZero function, this time expressed using nested types:

```
def incZero [s <: S] (x: s, y: s): s^t = x.inc (y.zero)
```

4 Classes

Based on our kernel language we now describe an object-oriented layer of classes. We do not treat classes as a new fundamental feature of the language, but see them as syntactic sugar, which can be defined in terms of the vase language. The subsequent sections will present a succession of refinements of the base class model and show how these can be represented in the class-less base language of nested types.

4.1 Simple Class Definitions

A class C introduces a type (of objects) C and a module with the same name. The module contains constructor functions for the objects described by the class. Following Palsberg and Schwarzbach [PS94] and [BOW98], we consider a set of classes that describe linked lists. as a running example for the material on classes.

We start with a simple class for linked lists, which are not meant to be refined further. This is expressed by the **final** modifier in the class definition.

```
final class LL (x: Int) = {
   private var next: LL
   def isEmpty: Boolean = x == –1
   def elem: Int = if (this.isEmpty) x else error();
   def getNext: LL = next
   def setNext(n: LL) = { next := n }
}
```

Linked lists have visible method isEmpty, elem, getNext and next. They also have a private field next which is can be accessed only locally.

The class above would translate to a type LL for linked-list objects, as well as a module LL which contains a constructor for such objects.

```
type LL = {
  def isEmpty: Boolean
  def elem: Int
  def getNext: LL
  def setNext(n: LL): ()
}
module LL = {
  def newInstance (x: Int) = {
    var next: LL
    def isEmpty: Boolean = x == −1
    def elem: Int = if (this.isEmpty) x else error();
    def getNext: LL = next
    def setNext(n: LL) = { next := n }
  }
}
```

The object type and the class module can be systematically derived from a class definition. Assume we have a class definition

final class $C(\tilde{x} : \tilde{T})$ = { ... }

Then the object-type is a record which contains bindings for every member of the class, except those members that have been marked with a **private** modifier. The corresponding module contains a single function newInstance, which takes the class parameters $C(\tilde{x} : \tilde{T})$ as parameters and the class body as body, with all **private** modifiers removed. That is, C has the outline:

module C = {
  def newInstance ($\tilde{x} : \tilde{T}$): C = { ... }
}

Note that this translation gives **private** members a weaker status than they have in Java or C++. In those languages, a private field of an object can be accessed from other objects of the same class. In our translation, a private field does not form part of the object type, so that it can only be accessed from the object in which it is defined. It would be possible to adapt our translation so that it more accurately reflects the standard meaning of **private** in C++ and Java. Essentially, the modified translation would use an abstract type for **private** members the implementation of which is known only within the containing class. While possible, this translation would share the shortcomings of the standard meaning of **private** in the area of scalability. For that reason, and because it is possible to emulate hiding via abstract types in user programs, we stick here with the "lexical closure" interpretation of **private**.

4.2 Static Class Members

Java allows class members to be marked **static**, to indicate that these members should exist once-per-class rather than once-per object. This is easily modelled by including such members as additional fields of a class module. As an example, assume we want to define an empty list in class LL:

```
final class LL (x: Int) = {
  static let empty = LL.newInstance (−1)
  ...
}
```

Then empty becomes another field of module LL. The type LL and the newInstance function remain as they were before.

```
module LL = {
  let empty = LL.newInstance (−1)
  def newInstance (x: Int) = {
    ...
  }
}
```

Note that by convention static members are defined before the newInstance function. Hence, they can refer to newInstance only indirectly, by selecting the module value itself.

4.3 Inheritance

So far, we considered a class exclusively as an object creator. We now extend this point of view by taking inheritance into account. A class can now serve as an inherited template that defines part of the behavior of objects belonging to subclasses. As an example, consider again class LL with an extension LLSub. LLSub objects are like LL objects, but the "emptiness" of a list object is determined by an additional class parameter empty instead of being encoded by the special value -1.

```
class LL(x: Int) = {
  private var next: LL
  def isEmpty: Boolean = x == −1
  def elem: Int = if (this.isEmpty) x else error()
  def getNext: LL = next
  def setNext(n: LL) = { next := n }
}
class LLSub(x: Int, empty: Boolean) = LL {
  inherit LL(x)
  def isEmpty = empty
}
```

These definitions make LLSub a subtype of LL. The definition of LLSub is based on the definition of LL. Instead of repeating the definition of members of LL in LLSub, we "inherit" those members using the clause **inherit** LL(x).

Following standard approaches, we express inheritance as *delegation*. The principle is that inherited methods are forwarded to methods of an object of the inherited class, the so-called delegate. As is customary, we use the name super for this object. Care is required with self references. We would expect the self reference this to refer to the inheriting object, not the delegate, so that overriding works: When calling this.isEmpty(...) in the LL delegate of a LLSub object, it should be the redefined version of isEmpty in LLSub that is called, not the original version of isEmpty as defined in LL. We achieve this by introducing a new object constructor function, newSuper, which takes the identity of the object being constructed as parameter.

Hence, a class definition **class** C = { ... } would give rise to a function

```
def newSuper (this: C): C { ... }
```

in module C. An inheriting class D would invoke this newSuper function to create a delegate object and forward all inherited functions to it.

```
val super = C.newSuper (this)
with super
```

The construct **with** E makes available all fields of the record E in with any qualification. If the term E is of a record type with fields x_1, ..., x_n, this term is equivalent to

val e = E ; **val** x_1 = e.x_1, ..., x_n = e.x_n.

for some fresh identifier e. To create a new C object, we need to invoke C.newSuper with the created object as parameter. Hence, C.newInstance becomes:

```
def newInstance = {
  val this = newSuper (this); this
}
```

The described technique implements the standard object-encoding via recursive records. Care is required in forming the right fixed point for this, since a pure call-by-value interpretation would lead to non-termination. We solve this in Funnel by evaluating recursive definitions like the one above in a lazy fashion. We first establish an as yet unfilled object for this, then pass this object to newSuper. When newSuper returns, all fields of the this object are overwritten with the corresponding fields of newSuper's result. During evaluation of newSuper it is quite legal to refer to this, for instance to pass it as an argument to another function. However, any attempt to dereference this will lead to a run-time error signalling a cyclic definition.

To return to our running example, here is the new translation of class LL:

```
type LL = {
  def isEmpty: Boolean
  def elem: Int
  def getNext: LL
  def setNext(n: LL): ()
}
module LL = {
  def newSuper (this: LL, x: Int): LL = {
    var next: LL
    def isEmpty: Boolean = x == −1
    def elem: Int = if (this.isEmpty) x else error()
    def getNext = next
    def setNext(n: LL) = { next := n }
  }
  def newInstance (x: Int): LL = { val this: LL = newSuper(this, x); this }
}
```

The translation of class LLSub shows the implementation of inheritance via delegation.

```
type LLSub = LL {}
module LLSub = {
  def newSuper (this: LLSub, x: Int, empty: Boolean): LLSub = {
    val super = LL.newSuper (this, x)
    with super
    def isEmpty: Boolean = empty
  }
  def newInstance (x: Int, empty: Boolean): LLSub = {
    val this: LLSub = newSuper(this, x, empty); this
  }
}
```

4.4 Abstract Methods

Assume now that we do not want to give an implementation of isEmpty at all, and instead want to defer such an implementation to subclasses. This can be expressed by an abstract method:

```
class LLAbs(x: Int) = {
  private var next: LL
  def getNext: LL = next
  def setNext(n: LL) = { next := n }
  abstract def isEmpty
  def elem = if (this.isEmpty) then x else error();
}
```

How should abstract methods be formalized? So far, the newSuper function returned a value of the same type as was passed into the function in the this parameter. With abstract methods, this changes. An abstract method still needs to be included in the type of the self reference, since it may be invoked as a method of this. However, it clearly cannot form part of the record defined by newSuper, since no definition is given. We model this by defining a new type named Trait in the class module. The Trait type collects all functions defined by that class. The object type is then composed of the trait type together with bindings for all abstract methods. Here are the details:

```
type LL = {
  with LL.Trait
  def isEmpty
}
module LL = {
  type Trait = {
    def elem: Int
    def getNext: LL
    def setNext(n: LL): ()
  }
  def newSuper (this: LL, x: Int): Trait = {
    var next: LL
    def getNext = next
    def setNext(n: LL) = { next := n }
    def elem = if (this.isEmpty) then x else error();
  }
}
```

Note that there is no newInstance function, which reflects the rule that no instances of classes with abstract members can be created. newInstance could not be defined anyway, since its fixed-point would not be type-correct. In

```
  val this: LL = newSuper (this, x)
```

the left-hand-side this has type LL, but the right-hand side's type is LL.Trait, which is not a subtype of LL.

4.5 Self-Types

So far, our object language's capabilities correspond roughly to those provided by common object-oriented languages such as Java or C#. We now discuss further refinements of this model, which go beyond the power of those languages, and which make essential use of the nested types concept.

Let's say we wish to extend class LL along another dimension, by defining a class DLL of doubly linked lists. Naturally, we would expect to be able to reuse the LL code in DLL through inheritance. However, with the translation established so far, this is not possible. After all, the new class DLL would have members:

```
private next: DLL
def getNext: DLL
def setNext(n: DLL): ()
```

With the subtyping rule for functions, which is contravariant in the argument type, we get that the type of LL.setNext is not a subtype of the type of DLL.setNext, hence the method override is illegal. The concept of *self-types* [Bru93] addresses this problem. The idea is to allow in a class definition references to the type This, which stands for the type of the object currently being defined. Hence, This might equal the enclosing class (if the object being defined is of the class itself) or a subclass (if the class is inherited). Using This, we can define singly and doubly linked lists as follows.

```
class LL(x: Int) = {
    private var next: This
    def elem: Int = x
    def getNext: This = next
    def setNext(n: This) = { next := n }
}
class DLL(x: Int) = LL {
    inherit LL(x)
    var prev: This
    def setNext(n: This) = { super.setNext(n) ; n.prev := this }
}
```

To reduce clutter, we drop from now on the isEmpty method, since it is inessential for the subsequent explanations. Note that the type of getNext is now the same in LL and DLL, hence it is possible to inherit getNext in DLL without modifications. Note also that type DLL is a subtype of LL; hence it is possible to pass a DLL object in all contexts where a LL object is specified. Therefore, the following function skipNext would work for singly as well as doubly linked lists.

```
def skipNext(xs: LL) = setNext(xs.getNext.getNext)
```

But for the same reason, the following code cannot be type correct:

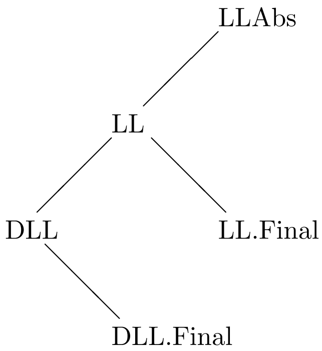
```
def crossLink(LL xs, LL ys) = { xs.setNext(ys); ys.setNext(xs) }
```

That code, if it passed the type checker, would be unsound. If crossLink was called with a LL object and a DLL object as parameters, one of the setNext operations would write to an non-existing field prev. However, an operation like crossLink can be applied to arguments if subtyping for these arguments is disallowed. One way to achieve this is by declaring a class **final**. The following version crossLink1 is type-correct:

```
final class FLL = LL {}
def crossLink1(FLL xs, FLL ys) = { xs.setNext(ys); ys.setNext(xs) }
```

The previous type soundness hole is now closed since FLL cannot be subtyped, so that the identity of This in FLL is fixed.

These observations can be generalized. The self-type technique rests essentially on two forms of types. There are class-types which can be subtyped, but which support only a restricted set of operations and there are final types which support the full set of operations but which should not be subtyped further. In our proposed class language this is made completely systematic by introducing automatically for every non-abstract and non-final class LL a "leaf" type LL.Final which extends the LL type and which fixes the identity of This. Hence, for a path DLL $\rightarrow$ LL $\rightarrow$ LLAbs in the class hierarchy we would get the following subtyping hierarchy of types:



How are self-types implemented? Let's first look at the role of This in type LL. Clearly, This stands for an unknown subtype of LL. So it makes sense to represent This as an abstract type component of LL:

type LL = {
    type This <: LL
    with LL.Trait[This]
}

As before there is a trait type LL.Trait which contains the methods of LL. Since the types of those methods may refer to This, we parameterize LL.Trait with This. The newSuper method in LL will now have the following signature:

def newSuper [This <: LL] (this: This, x: Int): Trait[This]

That is, newSuper takes a type parameter [This <: LL] representing the type of the object to be created, and it takes a value parameter (this: This) representing the object itself.

The last ingredient we need for the translation of class LL is the leaf type LL.Final, which fixes the meaning of This. That type is easily defined:

type Final = LL { **type** This = Final }

Figure 1 presents the complete translation of classes LL and DLL. With this translation in mind, it is now easy to determine the type-correctness of operations over the linked-list classes. Let's study skipNext first.

def skipNext(xs: LL) = xs.setNext(xs.getNext.getNext)

Given our typing rule for record selection, we obtain from the premise xs:LL:

xs.getNext	: xs.This
xs.getNext.getNext	: xs.This
xs.setNext	: xs.This $\rightarrow$ ()

```
type LL = { type This <: LL; with LL.Trait[This] }

module LL = {
    type Trait[This <: LL] = {
        def elem: Int
        def getNext: This
        def setNext(n: This): ()
    }
    def newSuper [This <: LL] (this: This, x: Int): Trait[This] = {
        var next: This
        def elem = x
        def getNext = next
        def setNext(n: This) = { next := n }
    }
    type Final = LL { type This = Final }
    def newInstance(x: Int): Final = {
        val this: Final = { with newSuper [This] (this, x) } ; this
    }
}

type DLL = { type This <: DLL; with DLL.Trait[This] }

module DLL = {
    type Trait[This <: DLL] = LL.Trait[This] {
        var prev: This
        def setNext(n: This): ()
    }
    def newSuper [This <: DLL] (this: This, x: Int): Trait[This] = {
        val super: LL.Trait[This] = LL.newSuper [This] (this, x)
        with super
        var prev: This
        def setNext(n: This) = { super.setNext(n); n.prev := this }
    }
    type Final = DLL { type This = Final }
    def newInstance(x: Int): Final = {
        val this: Final = { with newSuper [This] (this, x) } ; this
    }
}
```

Figure 1: Translation of classes LL and DLL

Hence, xs.setNext(xs.getNext.getNext): (). Turning to crossLink, we find:

```
xs.setNext        : xs.This → ()
ys:    : LL
```

Since LL is not a subtype of xs.This, we get a type error. On the other hand, the following version crossLink2 is type correct:

def crossLink2(LL.Final xs, LL.Final ys) = { xs.setNext(ys); ys.setNext(xs) }

To see this, observe that xs.setNext: xs.This → (). Since xs: LL.Final and This in LL.Final is a type alias for with LL.Final, the type of xs.setNext is equal to LL.Final → (). Since the ys parameter is of type LL.Final, the application xs.setNext(ys) is type correct.

However, the typability is paid for by a loss in flexibility. In particular it is not possible to apply crossLink2 to doubly linked lists because DLL is not a subtype of LL.Final. To make up for this loss, Kim Bruce replaces the subtyping relation between types with a *matching* relation. In our formalization, matching relates a class with all its leaf subtypes. In the case of crossLink2, we would like to express that this function can be applied to all leaf subtypes T of LL for which T.This = T holds.

Matching is not a primitive relation in our formalization, but it can be simulated using bounded quantification with a type equality constraint. Here is an example:

def [T <: LL { **type** This = T }] crossLink3 (xs: T, ys: T): ()

This makes crossLink3 applicable to all pairs of lists xs, ys, where both xs and ys are of some subtype T of LL which satisfies the constraint that T.This = T.

The relation of this formalization with the approach of Bruce can be summarized as follows: A type T in his formalization corresponds to T.Final in ours. His hash type #T corresponds to our T. The matching relation in his formalization is modelled here by bounded quantification with type equality constraints. Bruce uses only matching and foregoes subtyping completely. This seems to prevent some useful program examples. For example, the skipNext function given previously could not be applied to an argument of type DLL if function arguments are subject to just matching, but not subtyping.

4.6 Co-variant Families of Classes

Self-types allow a co-variant specialization of a type variable representing the type of the object being defined. They are by themselves not strong enough to model families of classes that are specialized together. Canonical examples of this are the subject-observer pattern [GHJV94] or the nodes and edges of a graph. Virtual types have been proposed as a way to express this form of polymorphism [MMP89, Tor98, IP99, BOW98].

Keeping with our running example of linked lists, the setting can be explained as follows. We wish to construct linked lists of *alternating* elements. There are now two node types, LLX and LLY. The elem method of an LLX object returns a value of type X whereas the same method in an LLY object returns a value of type Y. LLX objects refer via their next field to LLY objects and *vice versa*. So far, this is simple to express. But as in the previous section, we want to arrange for a later extension of classes LLX and LLY to alternating doubly-linked lists with classes DLLX and DLLY. In the non-alternating case, we required an abstract type This which referred to the unknown type of the object being defined. Let's now generalize this technique to two abstract types, one standing for the type of the object being defined, the other standing for its partner type in the alternating list, which will be named That:

```
class LLX(x: X) = {
    type That <: LLY { type That = This }
    private var next: That
    def elem: X = x
    def getNext: That = next
    def setNext(n: That) = { next := n }
}
class LLY(y: Y) = {
    type That <: LLX { type That = This }
    private var next: That
    def elem: Y = y
    def getNext: That = next
    def setNext(n: That) = { next := n }
}
```

Note the difference in the treatment of the names This and That. The name This is built into the class language. It is used as the type of this, which is also built on. On the other hand, That is explicitly defined and is used as the type of the explicitly defined instance variable next.

Classes LLX and LLY may be extended by a pair of subclasses, as long as the extending subclasses conform to each other. The conformance requirement for extending classes CX and CY is that

$$\mathsf{CX.That} = \mathsf{CY.This} \qquad \text{and} \qquad \mathsf{CY.That} = \mathsf{CX.This}.$$

For pairs of leaf types TX and TY we obtain:

$$\mathsf{TX.That} = \mathsf{TY} \qquad \text{and} \qquad \mathsf{TY.That} = \mathsf{TX}.$$

The conformance requirement is expressed in the type constraint

```
{ type That = This }
```

in the definitions of types That in both LLX and LLY. With this construction, it is now possible to extend the pair (LLX,LLY) to a pair of doubly linked list classes (DLLX,DLLY):

```
class DLLX(x: X) = LLX {
    inherit LLX(x)
    type That <: DLLY { type That = This }
    var prev: That
    def setNext(n: That) = { super.setNext(n) ; n.prev := this }
}
class DLLY(y: Y) = LLY {
    inherit LLY(x)
    type That <: DLLX { type That = This }
    var prev: That
    def setNext(n: That) = { super.setNext(n) ; n.prev := this }
}
```

It remains to determine how abstract types like That are handled in our class translation. The idea is to treat in a class definition every abstract type **type** T <: C {...} which is bounded by a single class type C in the same way we treated the This type in the previous sub-section. In particular, the Trait type is now parameterized by all such abstract types, and the Final type contains for every such type a type alias **type** T = C. This generalized translation no longer needs a special case for type This. Instead, This in class C is treated as if there was an implicit declaration **type** This <: C in C.

Figure 2 presents the complete translation of classes LLX and DLLX which follows these principles. The translation of LLY and DLLY is completely symmetric and has therefore been omitted.

```
type LLX = {
   type This <: LLX
   type That <: LLY {type That = This}
   with LLX.Trait[This]
}
module LLX = {
   type Trait[This <: LLX, That <: LLY {type That = This}] = {
      def elem: X
      def getNext: That
      def setNext(n: That): ()
   }
   def newSuper [This <: LLX, That <: LLY {type That = This}]
                         (this: This, x: Int): Trait[This] = {
      var next: That
      def elem: X = x
      def getNext: That = next
      def setNext(n: That) = { next := n }
   }
   type Final = LLX { type This = Final, type That = LLY.Final }
   def newInstance(x: Int): Final = {
      val this: Final = { with newSuper [This] (this, x) } ; this
   }
}
type DLLX = LLX {
   type This <: DLLX
   type That <: DLLY { type That = This }
   with DLLX.Trait[This]
}
module DLLX = {
   type Trait[This <: DLLX, That <: DLLY {type That = This}] = LLX.Trait {
      var prev: That
      def setNext(n: That)
   }
   def newSuper [This <: DLLX, That <: DLLY {type That = This}]]
                         (this: This, x: Int): Trait[This] = {
      val super: LLX.Trait[This] = LLX.newSuper [This] (this, x)
      with super
      var prev: That
      def setNext(n: That) = { super.setNext(n); n.prev := this }
   }
   type Final = DLLX { type This = Final, type That = DLLY.Final }
   def newInstance(x: Int): Final = {
      val this: Final = { with newSuper [This] (this, x) } ; this
   }
}
```

Figure 2: Translation of classes LLX and DLLX

Let's now look at the usage examples developed in the previous section in the context of alternating lists. First, here is the analogue of skipNext:

```
def skipNext2(xs: LLX) = { xs.setNext(xs.getNext.getNext.getNext) }
```

To demonstrate type-correctness, one remarks that:

```
xs.getNext          : xs.That
xs.getNext.getNext  : xs.That.That
```

The latter type equals xs.This, hence we also have

```
xs.getNext.getNext.getNext :    xs.That
```

which is what's required.

The definition of crossLink can also be adapted to alternating lists. Here is a version which works for arguments of arbitrary pairs of conforming subclasses of LLX and LLY:

```
def crossLink4 [X <: LLX {type That = Y}, Y <: LLY {type That = X}] (x: X, y: Y) = {
   x.setNext (y) ; y.setNext (x)
}
```

The last example is interesting since it shows that the nested types are sufficiently powerful to express the general sharing constraints of ML module systems. If LLX and LLY were module signatures, crossLink could be written in SML as a functor in the following way:

```
functor crossLink(x: LLX, y: LLY sharing x.That = y, y.That = x) = ...
```

SML's sharing constraints relate abstract type components of structures. By contrast, bounded quantification in nested types allows us to relate type variables of polymorphic functions. But, as the example shows, we can simulate an SML sharing constraint by declaring type variables for the constrained types and using bounded quantification on those type variables.

5 Conclusion

The last section has described a long succession, from simple class definitions to self-types and co-variant families of classes. Standard recursive records were sufficient to type basic class definitions, but nested types were essential to model the more advanced concepts. Nested types also support directly standard abstract data types and, with a little bit of encoding, SML-style functors. The main advantage of nested types is thus that they provide in a uniform type system the abstraction facilities of SML-style modules and the composition facilities of advanced object-oriented languages. Nested types seem therefore to be a good candidate for a type-systematic basis of languages that combine modules and objects in a unified way.

At present this work is preliminary at best. We are currently working towards a more detailed study of the meta-theory of nested types. In particular, work is under way to prove subject reduction and decidability of type checking. Another question worth further research are structural formulations of nested types, and their relationship to other type theories.

References

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, pages 1–13. Springer-Verlag, August 1991. Lecture Notes in Computer Science 528.

[AOS+00] Philippe Altherr, Martin Odersky, Michel Schinz, Matthias Zenger, and Christoph Zenger. Funnel distribution. available from `http://lampwww.epfl.ch/funnel`, July 2000.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A staticalle safe alternative to virtual types. In *Proc. 5th International Workshop on Foundations of Object-Oriented Languages, San Diego.*, January 1998.

[Bru93] Kim B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 285–298. ACM Press, January 1993.

[Car89] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, May 1989.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[CL90] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report report 56, DEC SRC, 1990.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, June 1997.

[Dug96] Dominic Duggan. Mixin modules. In *ACM SIGPLAN International Conference on Functional Programming*, 1996.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages (POPL)*, San Diego, January 1998. ACM.

[FR99] Kathleen Fisher and John Reppy. The design of a class mechanism for moby. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 37–49, 1999.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Java Series, Sun Microsystems, 1996. ISBN 0-201-63451-1.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the Twenty-First ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, pages 123–137, Portland, OR, January 1994.

[IP99] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *European Conference on Object-Oriented Programming (ECOOP).*, 1999. Also in informal proceedings of the Sixth International Workshop on Foundations of Object-Oriented Languages (FOOL).

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999.

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

[Ler94] Xavier Leroy. Manifest types, modules and separate compilation. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, Portland, OR, January 1994.

[Ler99] Xavier Leroy. *The Objective Caml system, release 2.04*. INRIA, 1999. available from `http://pauillac.inria.fr/ocaml/htmlman/`.

[Mac86] David B. MacQueen. Using dependent types to express modular structure. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, 1986.

[MMP89] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1989.

[MMPN93] O. Lehrmann Madsen, B. Mller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. ddison Wesley, June 1993.

[MP85] John C. Mitchell and Gordon C. Plotkin. Abstract types have existential type. In *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 37–51. Association for Computing Machinery, January 1985.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[NvO98] Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe — definitely. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 161–170, San Diego, January 1998. ACM.

[Ode00a] Martin Odersky. Functional nets. In *Proc. European Symposium on Programming*, number 1782 in LNCS, pages 1–25. Springer Verlag, March 2000.

[Ode00b] Martin Odersky. Objects + views = components? In *Abstract State Machines 2000*, LNCS. Springer Verlag, March 2000.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 146–159, Paris, France, January 1997.

[OZZ01] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, 2001.

[Pie92] Benjamin C. Pierce. Bounded quantification is undecidable. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 305–315, January 1992.

[PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type-Systems*. John Wiley, 1994.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, 1998.

[Rus98] Claudio Russo. *Types for Modules*. PhD thesis, University of Edinburgh, 1998.

[Tor98] Mads Torgersen. Virtual types are statically safe. In *Proceedings of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, San Diego, CA, January 1998.

A Typing Rules

This appendix presents complete typing rules for a nominal kernel type system with nested types. To keep the presentation compact, we have omitted the largely orthogonal issue of type parameterization.

The results about these typing rules are still preliminary. A proof of type soundness is currently under way, but has not yet been completed.

A.1 Syntax

We use the following alphabets:

Type-name	t, u
Term-name	x, y, z
Version	i, j, k

We distinguish between type names t and value names x. Both sorts of names can be decorated with version numbers i. These version numbers are needed to enable α-renaming of the names of record fields. Since field names act as binders in record definitions, they need to be α-rename in order to prevent name clashes during reduction. On the other hand, a field name serves as a fixed label for record selection. We solve the dilemma by decorating names with version numbers, e.g. x^i, t^j. Such names are called *versioned names*, or v-names for short. Only the version number, but not the proper name may be α-renamed. In other words, version numbers distinguis between logically different identifiers which have the same name. We don't expect that version numbers are written directly in user programs, and relegate this instead to the conceptual tasks of a compiler, the details of which are left unspecified here. Alternative treatments [HL94] replace version numbers by introducing two names per binding. One name would serve as a fixed record label, the other as an α-renamable internal reference.

The syntax of types, type environments and kinds is as follows.

Type	T, U	$::=$	$t^i \mid x^i! \mid T\hat{\ }t \mid T\hat{\ }x! \mid T \rightarrow U$
Bound	B	$::=$	$\epsilon \mid B, T$
Environment	Γ, Σ	$::=$	$\epsilon \mid \Gamma, t^i : \kappa \mid \Gamma, x^i : T$
Type Environment	Θ	$::=$	$\epsilon \mid \Theta, t^i : \kappa$
Kind	κ	$::=$	$< B\{\Sigma\} \mid = B\{\Sigma\} \mid \equiv T$

A type T or U is a type name with a version number, t^i, a qualified type $T\hat{\ }t$, a witness type $x^i!$ or $T\hat{\ }x!$, or a function type $T \rightarrow U$. We allow the contraction of "$!\hat{\ }$" to ".", e.g., $x!\hat{\ }t$ becomes $x.t$ and $T\hat{\ }x!\hat{\ }t$ becomes $T\hat{\ }x.t$.

A type bound B consists of a possibly empty sequence of types $T_1, \ldots, T_n$. Keeping with the nominal nature of the type system, there are no record types. Instead, a type name may be bound to a record signature by an environment. A type environment Θ is a sequence of bindings $t^i : \kappa$ of type names to kinds, separated by commas. An environment Γ or Σ contains in addition bindings $x^i : T$ of term-names to types. The domain $dom(\Gamma)$ is the set of v-names bound by it. The comma operator on environments or bounds is assumed to be commutative and associative, with the empty environment or bound ϵ as identity. Furthermore, we always require that the domains of two environments Γ, Σ adjoined with comma are disjoint.

A kind κ captures information about a bound type name. It can be of three forms. An alias kind $\equiv T$ indicates that the type name in question is equivalent to type T. A concrete definition kind $= B\{\Sigma\}$ indicates that the type name in question defines a new type with bounds B and definitions Σ. Finally, an abstract kind $< B\{\Sigma\}$ indicates that the type name in question is a subtype of all types in bound B and that has at least the definitions given in Σ. We let the $\leqq$ symbol range over either $<$ or $=$.

The syntax of terms is as follows.

Qualified name	X	$::=$	$x^i \mid X.x$
Value	V, W	$::=$	$X \mid D : T \mid \lambda x^i : T.E$
Expression	E, F	$::=$	$x^i \mid E.x \mid E(E') \mid \textbf{let } x^i = V \textbf{ in } E \mid \textbf{type } \Theta \textbf{ in } E$
Record	D	$::=$	$\epsilon \mid D, x^i = V$

Expressions E are constructed from single versioned names x^i, record selections $E.x$, applications $E(E')$, local value definitions **let** $x^i = V$ **in** E, and local type definitions **type** Θ **in** E. Qualified names X are a subset of expressions constructed from just versioned names and record selection. Separate from expressions is the syntactic class of values. Values V are constructed from qualified names, function abstractions $\lambda x^i : T.E$, and record values $D : T$, with D being a sequence of value definitions $x = V$.

Note that values are not themselves expressions, hence code like $f(\lambda x.x)$ is not syntactically correct. This restriction is necessary for the formulation and proof of subject-reduction. Without it, reduction could substitute a value V for a name x, which would take the witness type $x!$ to $V!$. But $V!$ is in general not a type; it is a type only if V is a qualified name. The restriction is easily overcome by introducing intermediate names for every value which is used as an expression and which is not already a qualified name. That is, V becomes **let** $x^i = V$ **in** x^i for some fresh name x^i. One can assume that a compiler would apply this transformation automatically prior to type-checking.

A.2 Relationship to Source Language

The type syntax presented here covers only a subset of the type constructions introduced in previous sections. We have omitted parameterized types and polymorphic functions. We have also omitted the issue of inheritance. Hence, unlike in the Funnel source language a type T defined as **type** $\mathsf{T} = \mathsf{U}$ { ... } needs to re-define or strengthen all definitions of U in its body {...}; it is not permissible to inherit these definitions form the supertype U. The source language allows such re-definitions but does not require them. There are some small syntactic changes: (<:) in the source language becomes (<) in the type system. When used for type aliasing, (=) in the source language becomes ($\equiv$). Type definitions in the source language are represented by environments in the type system. A binding **type** t κ in the source language is written $t : \kappa$ in the type system.

The term syntax covers only the subset of Funnel necessary to exhibit the essential type system issues. We can safely omit side-effects and concurrency because they have no effect on the type-systematic issues. We have also replaced the more flexible function and record definition syntax of Funnel by more rigid constructs for introducing λ-abstractions and records composed of values. However, it is still straightforward to encode purely-functional Funnel programs in this more stylized syntax by introducing names for intermediate results and using function application as a substitute for general let-bindings.

A.3 Judgements for Types

We now present typing rules for the following judgements which relate types, kinds, and environments.

$\Gamma \vdash T : \kappa$	Type T has kind κ.
$\Gamma \vdash T : \Sigma$	Type T defines environment Σ
$\Gamma \vdash T \equiv U$	Types T and U are equivalent.
$\Gamma \vdash T \leq U$	Type T is a subtype of type U.

Some of the typing rules make use of a type prefix operator ($T\hat{}$) for environments. Roughly, the environment $T\hat{}\Sigma$ equals Σ with all references to names in $dom(\Sigma)$ prefixed by $T\hat{}$. Formally,

$$T\hat{}\Sigma \quad = \quad (T\hat{})_\Sigma \Sigma'$$

where Σ' is a suitably α-renamed version of Σ and the mapping $(T\hat{})_\Sigma$ is defined on qualified type names as follows.

$$\begin{array}{lcll} (T\hat{})_\Sigma U & = & T\hat{}t\hat{}U & \text{if } T = t^i\hat{}U \text{ for some } t^i \in dom(\Sigma), U. \\ & & T\hat{}x.U & \text{if } T = x^i.U \text{ for some } x^i \in dom(\Sigma), U. \\ & & U & \text{otherwise} \end{array}$$

The mapping is extended homomorphically to function types and environments. When writing $(T\hat{\ })_\Sigma \Sigma'$ we assume that Σ' is suitably α-renamed so that it does not bind the leading name of T. As before, we allow the contraction of $x!\hat{\ }\Sigma$ to $x.\Sigma$.

$\boxed{\Gamma \vdash T : \kappa}$

$$(\textsc{Tid})\ \frac{t^i : \kappa \in \Gamma}{\Gamma \vdash t^i : \kappa} \qquad (\textsc{Tsel})\ \frac{\Gamma \vdash T : \Sigma \qquad t^i : \kappa \in T\hat{\ }\Sigma}{\Gamma \vdash T\hat{\ }t : \kappa}$$

$$(\textsc{Xid})\ \frac{x^i : T \in \Gamma \qquad \Gamma \vdash T : \Sigma}{\Gamma \vdash x^i! :< T\{\Sigma\}} \quad (\textsc{Xsel})\ \frac{\Gamma \vdash T : \Sigma \qquad x^i : U \in T\hat{\ }\Sigma \qquad \Gamma \vdash U : \Sigma'}{\Gamma \vdash T\hat{\ }x! :< U\{\Sigma'\}}$$

$\boxed{\Gamma \vdash T : \Sigma}$

$$(\textsc{GenTy})\ \frac{\Gamma \vdash T :\ \leqq B\{\Sigma\}}{\Gamma \vdash T : \Sigma} \qquad (\textsc{Alias})\ \frac{\Gamma \vdash T : \equiv U \qquad \Gamma \vdash U : \Sigma}{\Gamma \vdash T : \Sigma}$$

$$(\textsc{Arrow})\ \ \Gamma \vdash T \to U : \{(\cdot) : T \to U\}$$

The last rule associates with a function type $T \to U$ the environment $\{(\cdot) : T \to U\}$, where $(\cdot)$ is a reserved name standing for the function application operation. This "detour" smoothes the treatment of witness types of function identifiers. If $x : T \to U$, then $x!$ also produces the environment $\{(\cdot) : T \to U\}$. Rule (Apply) for function application requires its first operand to produce an environment of this form, and hence can be applied to expressions of functional witness types. A programming language might choose to make $(\cdot)$ user-definable, which would enable a convenient functional access syntax for user-defined types such as arrays of hash-tables.

$\boxed{\Gamma \vdash T \equiv T'}$

$$(\textsc{Refl}\equiv)\ \ \Gamma \vdash T \equiv T \qquad (\textsc{Sym}\equiv)\ \frac{\Gamma \vdash T \equiv T'}{\Gamma \vdash T' \equiv T}$$

$$(\textsc{Trans}\equiv)\ \frac{\Gamma \vdash T \equiv T' \qquad \Gamma \vdash T' \equiv T''}{\Gamma \vdash T \equiv T''} \qquad (\textsc{Taut}\equiv)\ \frac{\Gamma \vdash T : \equiv U}{\Gamma \vdash T \equiv U}$$

$$(\textsc{Tsel}\equiv)\ \frac{\Gamma \vdash T \equiv T'}{\Gamma \vdash T\hat{\ }t \equiv T'\hat{\ }t} \qquad (\textsc{Arrow}\equiv)\ \frac{\Gamma \vdash T \equiv T' \qquad \Gamma \vdash U \equiv U'}{\Gamma \vdash T \to U \equiv T' \to U'}$$

$$(\textsc{Tsel}\equiv\text{'})\ \frac{\Gamma \vdash T \leq U \qquad \Gamma \vdash U : \Sigma \qquad t^i \in dom(\Sigma) \backslash idts(\Sigma)}{\Gamma \vdash T\hat{\ }t \equiv U\hat{\ }t}$$

Rule (Tsel≡') allows one to equate a type component of a type with the same type component in a supertype, provided the type component is completely defined in the supertype. "Completely defined" means in this context that a type has a concrete definition $= B\{\Sigma\}$, and that bound B and definitions Σ also only refer to completely defined types in the supertype. If a type name is not completely defined, we say it is *instance dependent*. The set of instance-dependent type names $idts(\Sigma)$ of an environment Σ is defined to be the smallest set S which contains all type names t^i satisfying at least one of the following conditions:

– t^i :< $B\{\Sigma'\}$ in Σ, for some B and Σ',

– $t^i : \kappa$ in Σ and κ refers to a type name $u \in S$,

– $t^i : \kappa$ in Σ and κ refers to a value name $x^j \in dom(\Sigma)$.

$\boxed{\Gamma \vdash T \leq T'}$

$$(\textsc{Refl}\leq)\ \frac{\Gamma \vdash T \equiv T'}{\Gamma \vdash T \leq T'} \qquad (\textsc{Trans}\leq)\ \frac{\Gamma \vdash T \leq T' \qquad \Gamma \vdash T' \leq T''}{\Gamma \vdash T \leq T''}$$

$$(\textsc{Arrow}\leq)\ \frac{\Gamma \vdash T' \leq T \qquad \Gamma \vdash U \leq U'}{\Gamma \vdash T \to U \leq T' \to U'} \qquad (\textsc{Tsel}\leq)\ \frac{\Gamma \vdash T \leq U}{\Gamma \vdash T\hat{\ }t \leq U\hat{\ }t}$$

$$(\textsc{Taut}\leq)\ \frac{\Gamma \vdash T :\ \leqq U_1, \ldots, U_n\{\Sigma\}}{\Gamma \vdash T \leq U_i}$$

A.4 Judgements for Terms

The next two forms of judgements assign a type to an expression E and to a value V. Typing rules for these judgements are as follows.

$\boxed{\Gamma \vdash E : T}$

$$(\textsc{Var})\ \frac{x^i : T \in \Gamma}{\Gamma \vdash x^i : x^i!} \qquad (\textsc{Equiv})\ \frac{\Gamma \vdash E : T \qquad \Gamma \vdash T \equiv U}{\Gamma \vdash E : U}$$

$$(\textsc{Sel})\ \frac{\Gamma \vdash E : T \qquad \Gamma \vdash E : \Sigma \qquad x^i : U \in \Sigma}{\Gamma \vdash E.x : T\hat{\ }x!}$$

$$(\textsc{Apply})\ \frac{\Gamma \vdash E : T \qquad \Gamma \vdash T : \{(\cdot) : U' \to U\} \qquad \Gamma \vdash E' : T' \qquad \Gamma \vdash T' \leq U'}{\Gamma \vdash E(E') : U}$$

$$(\textsc{Let})\ \frac{\Gamma \vdash V : U \qquad \Gamma, x^i : U \vdash E : T \qquad x^i \notin fn(T, U)}{\Gamma \vdash \mathbf{let}\ x^i = V\ \mathbf{in}\ E : T}$$

$$(\textsc{TDef})\ \frac{\Gamma, \Theta \vdash \Theta\ wf \qquad \Gamma, \Theta \vdash E : U \qquad dom(\Theta) \cap fn(U) = \emptyset}{\Gamma \vdash \mathbf{type}\ \Theta\ \mathbf{in}\ E : U}$$

$\boxed{\Gamma \vdash V : T}$

$$(\textsc{Record})\ \frac{\Gamma \vdash T : \Sigma, \Theta \qquad \Gamma, \Sigma \vdash D : \Sigma}{\Gamma \vdash (D : T) : T}$$

$$(\textsc{Lambda})\ \frac{\Gamma \vdash T\ \mathit{wf} \qquad \Gamma, x^i : T \vdash E : U \qquad x^i \notin \mathit{fn}(T, U)}{\Gamma \vdash \lambda x^i : T.E : T \to U}$$

Qualified names X are typed as values in the same way as they are typed as expressions. Rule ($\textsc{Record}$) makes use of an auxiliary judgement $\Gamma \vdash D : \Sigma$ which states that the record definitions D generate the environment Σ. Typing rules for this judgement are:

$\boxed{\Gamma \vdash D : \Sigma}$

$$(\textsc{EmptyDef})\ \Gamma \vdash \epsilon : \epsilon \qquad (\textsc{Def})\ \frac{\Gamma \vdash D : \Sigma \qquad \Gamma \vdash V : T' \qquad \Gamma \vdash T' \leq T}{\Gamma \vdash D, x^i = V : \Sigma, x^i : T}$$

A.5 The Well-Formedness Criterion

The final set of typing rules determines whether a type environment is well-formed. In particular one needs to check that type aliases and subtypes are non-cyclic, that a subtype contains all definitions of its supertypes, and that corresponding definitions are compatible.

The following forms of judgements are used for this task.

$\Gamma \vdash \Sigma\ \mathit{wf}$	Environment Σ is well-formed within Γ.
$\Gamma \vdash T\ \mathit{wf}$	Type T is well formed within Γ.
$T\,;\Gamma \vdash \Sigma\ \mathit{wf}$	Environment Σ which represents the definitions of type T within Σ is well-formed.
$T\,;\Gamma \vdash \kappa\ \mathit{wf}$	Kind κ which represents the kind of type T within Σ is well-formed.
$T\,;\Gamma \vdash \Sigma \leq \Sigma'$	Environment Σ representing the definitions of type T within Σ subsumes Σ'.
$T\,;\Gamma \vdash \kappa \leq \kappa'$	Kind κ of type T within Γ subsumes kind κ'.

Some technical difficulty arises from the fact that a type may be referred to by a local as well as a global name. For instance, consider the declarations

```
type t = {
   type u = {}
   ...
}
val x: t
```

at the point ... the names x.u, t\sel u and u all refer to the same type. This is reflected by passing the type name t into the judgement which checks the well-formedness of the definition of u.

$\boxed{\Gamma \vdash T\ \mathit{wf}}$

$$(\textsc{GenTy-wf})\ \frac{\Gamma \vdash T : \leqq B\{\Sigma\} \qquad \Gamma \vdash B\ \mathit{wf}}{\Gamma \vdash T\ \mathit{wf}}$$

$$(\textsc{Alias-wf})\ \frac{\Gamma \vdash T :\equiv U \qquad \Gamma \vdash U\ \mathit{wf}}{\Gamma \vdash T\ \mathit{wf}} \qquad (\textsc{Arrow-wf})\ \frac{\Gamma \vdash T\ \mathit{wf} \qquad \Gamma \vdash U\ \mathit{wf}}{\Gamma \vdash T \to U\ \mathit{wf}}$$

$\boxed{\Gamma \vdash \Sigma\ \mathit{wf}}$

$$(\textsc{Empty-wf})\ \ \Gamma \vdash \epsilon\ \mathit{wf} \qquad (\textsc{TBind-wf})\ \frac{\Gamma \vdash \Sigma\ \mathit{wf} \qquad t^i\,;\Gamma \vdash \kappa\ \mathit{wf}}{\Gamma \vdash \Sigma, t^i : \kappa\ \mathit{wf}}$$

$$(\textsc{XBind-wf})\ \frac{\Gamma \vdash \Sigma\ \mathit{wf} \qquad \Gamma \vdash U\ \mathit{wf}}{\Gamma \vdash \Sigma, x^i : U\ \mathit{wf}}$$

$\boxed{T\,;\Gamma \vdash \Sigma\ \mathit{wf}}$

$$(\textsc{Empty-wf'})\ \ T\,;\Gamma \vdash \epsilon\ \mathit{wf} \qquad (\textsc{TBind-wf'})\ \frac{T\,;\Gamma \vdash \Sigma\ \mathit{wf} \qquad T\hat{}t\,;\Gamma \vdash \kappa\ \mathit{wf}}{T\,;\Gamma \vdash \Sigma, t^i : \kappa\ \mathit{wf}}$$

$$(\textsc{XBind-wf'})\ \frac{T\,;\Gamma \vdash \Sigma\ \mathit{wf} \qquad \Gamma \vdash U\ \mathit{wf}}{T\,;\Gamma \vdash \Sigma, x^i : U\ \mathit{wf}}$$

$\boxed{T\,;\Gamma \vdash \kappa\ \mathit{wf}}$

$$(\textsc{GenTy-wf'})\ \frac{\begin{array}{c} T\,;\Gamma, x^i : T \vdash x^i.\Sigma\ \mathit{wf} \qquad \Gamma \vdash U_i : \Sigma_i \\ T\,;\Gamma, x^i : T \vdash x^i.\Sigma \le x^i.\Sigma_j \quad (j = 1, ..., n) \end{array}}{T\,;\Gamma \vdash (\leqq U_1, \ldots, U_n\{\Sigma\})\ \mathit{wf}}$$

$$(\textsc{Alias-wf'})\ \frac{\Gamma \vdash U\ \mathit{wf}}{T\,;\Gamma \vdash (\equiv U)\ \mathit{wf}}$$

Note that in rule (GenTy-wf') the name x^i is arbitrary, but is not allowed to be in the domain of Γ. Intuitively x^i is a placeholder name which stands for an arbitrary, but fixed implementation of Σ. An analogous scheme is used in rule (GenTy$\le$).

$\boxed{T\,;\Gamma \vdash \Sigma \le \Sigma'}$

$$(\textsc{Empty}\le)\ \ T\,;\Gamma \vdash \Sigma \le \epsilon \qquad (\textsc{TBind}\le)\ \frac{T\,;\Gamma \vdash \Sigma \le \Sigma' \qquad T\hat{}t\,;\Gamma \vdash \kappa \le \kappa'}{T\,;\Gamma \vdash \Sigma, t^i : \kappa \ \le\ \Sigma', t^j : \kappa'}$$

$$(\textsc{XBind}\le)\ \frac{T\,;\Gamma \vdash \Sigma \le \Sigma' \qquad \Gamma \vdash U \le U'}{T\,;\Gamma \vdash \Sigma, x^i : U \ \le\ \Sigma', x^j : U'}$$

$\boxed{T\,;\Gamma \vdash \kappa \le \kappa'}$

$$(\textsc{Alias}\le)\ \frac{\Gamma \vdash U : \kappa}{T\,;\Gamma \vdash (\equiv U) \le \kappa} \qquad (\le\textsc{Alias})\ \frac{\Gamma \vdash U : \kappa}{T\,;\Gamma \vdash \kappa \le (\equiv U)}$$

$$(\textsc{Trans}\le)\ \frac{T\,;\Gamma \vdash \kappa \le \kappa' \qquad T\,;\Gamma \vdash \kappa' \le \kappa''}{T\,;\Gamma \vdash \kappa \le \kappa''}$$

$$(\textsc{GenTy}\le)\ \frac{\forall j.\exists i.\quad \Gamma \vdash U_i \le U'_j \qquad T\,;\Gamma, x^i : T \vdash x^i.\Sigma \le x^i.\Sigma'}{T\,;\Gamma \vdash (\leqq U_1, \ldots, U_m\{\Sigma\}) \le (< U'_1, \ldots, U'_n\{\Sigma'\})}$$

A.6 Reduction Semantics

In the following we define a reduction relation $\hookrightarrow$. If we used straightforward β-reduction on terms, a substitution-lemma and therefore subject reduction would not hold. In order to show a subject reduction result we give a new name to each intermediate result during reduction. The definitions for those new names are collected in the record D, their types in the environment Γ. The reduction is then defined on triples $[\Gamma\,;D \vdash E]$.

$\boxed{[\Gamma\,;D \vdash E] \hookrightarrow [\Gamma'\,;D' \vdash E]}$

$$(\textsc{App1})\ \frac{[\Gamma\,;D \vdash E] \hookrightarrow [\Gamma'\,;D' \vdash E']}{[\Gamma\,;D \vdash E(F)] \hookrightarrow [\Gamma'\,;D' \vdash E'(F)]}$$

$$(\textsc{App2})\ \frac{[\Gamma\,;D \vdash E] \hookrightarrow [\Gamma'\,;D' \vdash E']}{[\Gamma\,;D \vdash X(E)] \hookrightarrow [\Gamma'\,;D' \vdash X(E')]}$$

$$(\textsc{App3})\ \frac{D \vdash X = \lambda x^i : T.E \qquad \Gamma \vdash Y : T' \qquad \Gamma \vdash T' \leq T}{[\Gamma\,;D \vdash X(Y)] \hookrightarrow [\Gamma\,;D \vdash [Y/x^i]E]}$$

$$(\textsc{Sel})\ \frac{[\Gamma\,;D \vdash E] \hookrightarrow [\Gamma'\,;D' \vdash E']}{[\Gamma\,;D \vdash E.x] \hookrightarrow [\Gamma'\,;D' \vdash E'.x]} \quad (\textsc{Let})\ \frac{\Gamma \vdash V : T}{[\Gamma\,;D \vdash \mathbf{let}\ x^i = V\ \mathbf{in}\ F] \hookrightarrow [\Gamma, x^i : T\,;D, x^i = V \vdash F]}$$

$$(\textsc{Type})\ \frac{}{[\Gamma\,;D \vdash \mathbf{type}\ \Sigma\ \mathbf{in}\ E] \hookrightarrow [\Gamma, \Sigma\,;D \vdash E]}$$

(App1) and (App2) assure that we reduce function and argument before the application. For the application (App3) we look up the value V for X in the record D

$\boxed{D \vdash X = V}$

$$(\textsc{Simp})\ \ D, x^i = V \vdash x^i = V \qquad\qquad (\textsc{Sel})\ \frac{D \vdash X = (D' : T) \qquad D' \vdash x^i = V}{D \vdash X.x = (X\,\hat{}\,)_{D'}V}$$

$$(\textsc{Fwd})\ \frac{D \vdash X = Y \qquad D \vdash Y = V}{D \vdash X = V}$$

Here $(X\,\hat{}\,)_{D'}$ is defined analogous to $(X\,\hat{}\,)_{\Sigma}$. Since an expression E always reduces to a qualified name X and never to a value V, we are only substituting qualified names for names. This makes a substitution-lemma possible.

(Let) and (Type) simply put the new value or type in the environment. The properties of the comma operator ensure that we choose a new name for each binding.

A.7 Subject Reduction

Subject reduction is described by two invariants of every reduction sequence. First, the expression E has to be typable given the environment Γ, where the type of E may get smaller but not bigger during reduction.

Second, for each triple $[\Gamma\,;D \vdash E]$ of the reduction sequence all the definitions in D are type correct with respect to Γ. This is expressed by

$\boxed{\Gamma \vdash D}$

$$(\textsc{Empty})\ \Gamma \vdash \epsilon \qquad (\textsc{Comma})\ \frac{\Gamma \vdash D \qquad \Gamma \vdash V : T \qquad \Gamma \vdash T \leq T' \qquad (x^i : T') \in \Gamma}{\Gamma \vdash D, x^i = V}$$

Conjecture **[Subject Reduction]** If $\Gamma \vdash E : T$ and $\Gamma \vdash D$ and $[\Gamma ; D \vdash E] \hookrightarrow [\Gamma, \Gamma' ; D, D' \vdash E']$ then T' exists with $\Gamma, \Gamma' \vdash E' : T'$ and $\Gamma, \Gamma' \vdash T' \leq T$ and $\Gamma, \Gamma' \vdash D, D'$.