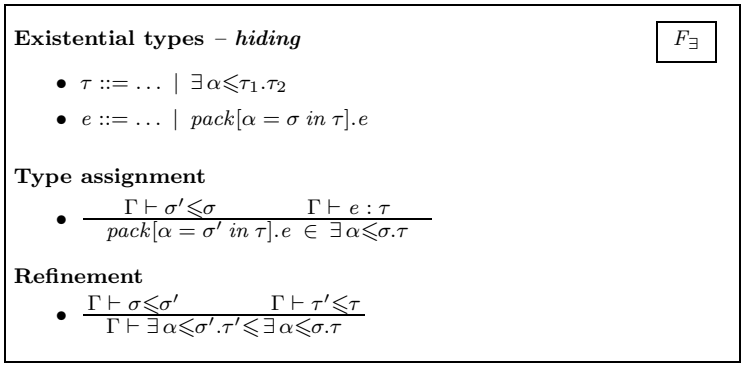
0.1 Existential types – hiding

Existential types were introduced in [CW85] to model aspects of data abstraction and hiding. The language introduced in [CW85] is essentially a variant of the typed lambda calculi we have looked at previously.

Our new calculus, that we call $F_{\exists}$, is an extension of $F_{\leqslant}$ with type expressions of the form $\exists\,\alpha{\leqslant}\sigma.\tau$ (to denote existential types) and expressions of the form $pack[\alpha = \sigma\ in\ \tau]$ (to denote values with hidden types). Intuitively, the meaning of the expression $pack[\alpha = \sigma\ in\ \tau]$ is that we represent the abstract type α occurring in the type expression τ by the actual type σ (in order to realize the value e). Following the type assignment rule, we may actually provide an instance of a subtype of the bounding type as the realization of a hidden type. See slide **??**.



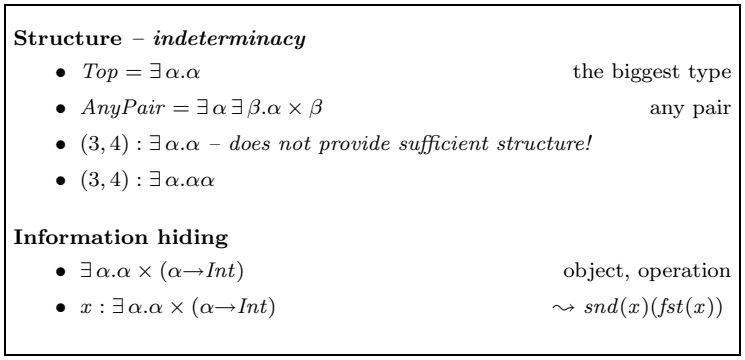
0-1

Slide 0-1: The existential type calculus

The subtyping refinement rule is similar to the refinement rule for universally quantified types. Notice also here the contravariance relation between the bounding types.

More interesting is what bounding types allow us to express. (As before, we will write $\exists\,\alpha.\tau$ to denote $\exists\,\alpha{\leqslant}\mathit{Top}.\tau$.) First, existential types allow us to indicate that the realization of a particular type exists, even if we do not indicate how. The declaration $e : \exists\,\alpha.\tau$ tells us that there must be some type σ such that e of type τ can be realized. Apart from claiming that a particular type exists, we may also provide information concerning its structure, while leaving its actual type undetermined.

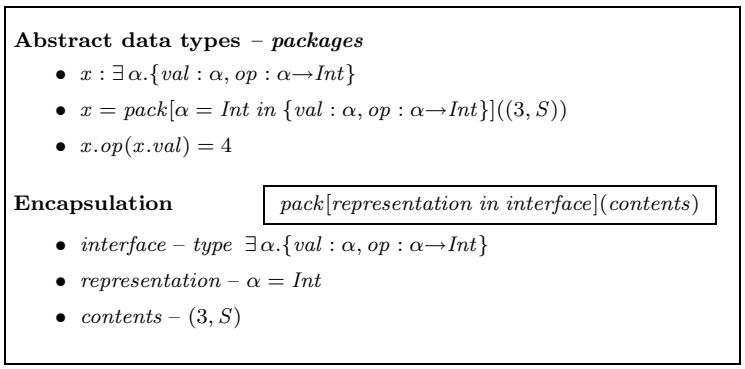
For example, the type $\exists\,\alpha.\alpha$ (which may clearly be realized by any type) carries no information whatsoever, hence it may be considered to be equal to the type *Top*. More information, for example, is provided by the type $\exists\,\alpha\,\exists\,\beta.\alpha \times \beta$ which defines the product type consisting of two (possibly distinct) types. (A product may be regarded as an unlabeled record.) The type $\exists\,\alpha.\alpha \times \alpha$ gives even more information concerning the structure of a product type, namely that the two components are of the same type. Hence, for the actual product $(3, 4)$ the latter is the best choice. See slide **??**.



Slide 0-2: Existential types – examples

Existential types may be used to impose structure on the contents of a value, while hiding its actual representation. For example, when we have a variable x of which we know that it has type $\exists\,\alpha.\alpha \times (\alpha \rightarrow Int)$ then we may use the second component of x to produce an integer value from its first component, by $snd(x)(fst(x))$, where *fst* extracts the first and *snd* the second component of a product. Clearly, we do not need to know the actual representation type for α.

A similar idea may be employed for (labeled) records. For example, when we have a record x of type $\exists\,\alpha.\{val : \alpha, op : \alpha \rightarrow Int\}$ then we may use the expression $x.op(x.val)$ to apply the operation *op* to the value *val*. Again, no knowledge of the type of *val* is required in this case. However, to be able to use an element of an existential type we must provide an actual representation type, by instantiating the type parameter in a *pack* statement.



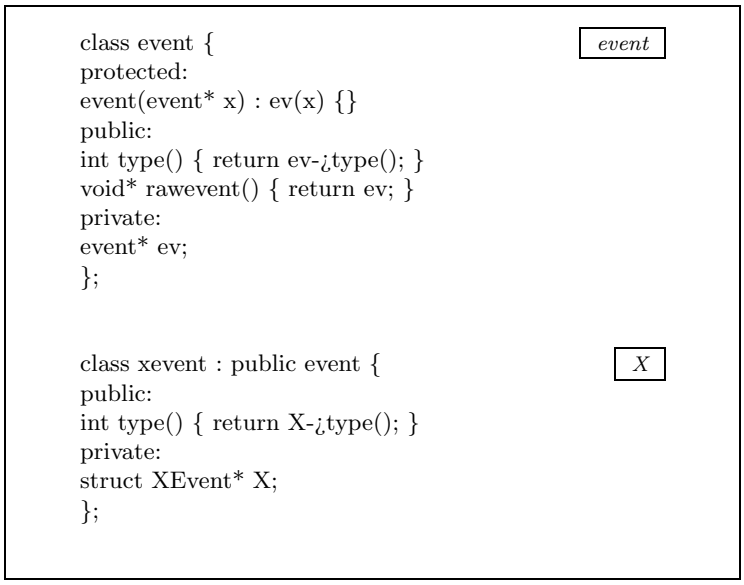
Slide 0-3: Packages – examples

The *pack* statement may be regarded as an encapsulation construct, allowing us to protect the inner parts of an abstract data type. When we look more closely at the *pack* statement, we can see three components. First, we have an

interface specification corresponding to the existential type associated with the *pack* expression. Secondly, we need to provide an actual representation of the hidden type, *Int* in the example above. And finally, we need to provide the actual contents of the structure. See slide **??**.

In combination with the notion of *objects as records*, existential types provide us with a model of abstract data types. Real objects, however, require a notion of *self-reference* that we have not captured yet. In the next section we will conclude our exploration of type theories by discussing the F_μ calculus that supports recursive (object) types and inheritance.

Hiding in C++ Naturally, the classical way of data hiding in C++ is to employ *private* or *protected* access protection. Nevertheless, an equally important means is to employ an abstract interface class in combination with forwarding.



Slide 0-4: Hiding in C++

For example, as depicted in slide **??**, we may offer the user a class *event* which records information concerning events occurring in a window environment, while hiding completely the underlying implementation. The actual *xevent* class realizing the type *event* may itself need access to other structures, as for example those provided by the X window environment. Yet, the *xevent* class itself may remain entirely hidden from the user, since events are not something created directly (note the protected constructor) but only indirectly, generally by the system in response to some action by the user.