# Typed Compilation Against Non-Manifest Base Classes

Christopher League[1] and Stefan Monnier[2]

[1] Long Island University
christopher.league@liu.edu
[2] Université de Montréal
monnier@iro.umontreal.ca

**Abstract.** Much recent work on proof-carrying code aims to build certifying compilers for single-inheritance object-oriented languages, such as Java or C#. Some advanced object-oriented languages support compiling a derived class without complete information about its base class. This strategy—though necessary for supporting features such as mixins, traits, and first-class classes—is not well-supported by existing typed intermediate languages. We present a low-level IL with a type system based on the Calculus of Inductive Constructions. It is an appropriate target for efficient, type-preserving compilation of various forms of inheritance, even when the base class is unknown at compile time. Languages (such as Java) that do not require such flexibility are not penalized for it at run time.

## 1 Motivation

In most object-oriented languages, programmers factor their solutions over a hierarchy of *classes*. Since the classes in a hierarchy may appear in different compilation units, one question that the language designer (or implementer) must address is: how much information about a base class is needed to compile its derived class?

With its emphasis on efficient object layout and method dispatch, C++ [32] requires *complete* information about the base class: the number, locations, and types of all its fields and methods. Indeed, it is because C++ depends on this information that a seemingly minor change to a base class triggers recompilation of all its descendents. Java [23] is somewhat more flexible. To support binary compatibility, its class files are not committed to a particular object layout. A derived class depends only on the names and types of the base class fields and methods that it uses. Nevertheless, most Java implementations ultimately compile classes to lower-level code using the same layouts and techniques as C++.

A few modern object-oriented languages allow classes as module parameters (Moby [15], OCaml [28]) or as first-class values (Loom [5]). Other languages support more flexible forms of inheritance, such as mixins [24, 3] and traits [29]. If a base class is not available for inspection when a derived class is compiled, we say the base class is not *manifest*. Implementations of these languages use a *dictionary* data structure to map method and field names to their locations in the object layout. The dictionary may be applied at link time or at run time, as required by the language.

Here is a simple example in OCaml (although it could be expressed just as easily in Moby). We declare a signature for modules containing a `circle` class that implements



three methods: `center`, `radius`, and `area`. The abstract type `spec` permits different implementations of this signature to have different constructor arguments.

```
module type CIRCLE =
sig type spec
    class circle : spec -> object
       method center : float*float
       method radius : float
       method area   : float
    end
end
```

Below, `CircleBBox` declares a class `bbox` that inherits from a (non-manifest) base class `circle`, overrides the `area` method (using a super call), and defines a new method `bounds`.

```
module CircleBBox = functor (C : CIRCLE) -> struct
   class bbox arg = object (self)
      inherit C.circle arg as super
      method area = super#area * 4.0 / pi (* area of bbox *)
      method bounds = let (x,y) = self#center in
                      let r = self#radius in
                      ((x-r,y-r), (x+r,y+r))
   end
end
```

To compile this functor, we must make do with relatively little information about the super class. We know it has the three methods specified in the signature, but not their positions nor whether there are other (hidden) methods, nor even the size of objects. We will return to this example throughout the paper.

Designing an effective *intermediate language* (IL) for compilers of these languages is challenging. Although method invocation is atomic at the source level, the IL should explicitly represent the dictionary search, method dereference, and (indirect) function call as separate operations. This way the operations may be independently optimized: combined, inlined, eliminated, or hoisted out of loops. To support such optimizations, Fisher, Reppy, and Riecke designed Links, a calculus for compiling and linking classes, based on the untyped λ-calculus. Its primitives can be combined "to express a wide range of class-based object-oriented features, such as class construction and various forms of method dispatch." [17]

In recent years, many researchers have based intermediate languages on *typed* λ-calculi. In addition to supporting type-directed optimizations, typed ILs are suitable for generating certified object code, such as typed assembly language [25] or proof-carrying code [26, 1]. Colby et al. [9] and League et al. [21, 22] have developed certifying compilers for Java, but more advanced class mechanisms are not yet well supported in this arena.

This paper presents a new intermediate language based on Links, but with a sound and decidable type system. We adopt the 'certified binaries' framework of Shao et al.

$$e ::= x \mid n \mid e_1 + e_2 \mid \lambda x.e \mid e_1\ e_2 \mid \langle e_1, ..., e_n \rangle \mid e_1 @ e_2 \mid e_1 @ e_2 \leftarrow e_3$$
$$\mid e \wr \langle e_1, ..., e_n \rangle \mid \{l_1 = e_1, ..., l_n = e_n\} \mid e \# l$$

**Fig. 1.** Links expression syntax.

[30], in which the types and proofs that govern computations are defined within the Calculus of Inductive Constructions [11, 12]. Our language has the same primitive operators as Links, so it is an appropriate target for efficient, type-preserving compilation of various forms of inheritance, even when the base class is unknown at compile time. Moreover, languages (such as Java) that do not require such flexibility are not penalized for it at run time.

In the next section, we review the primitives of Links and explain an untyped translation of our running example. Section 3 introduces the framework of our type language, and develops the semantics of LITL, our computation language. We revisit the example, now in a typed setting, in section 4. Section 5 explores techniques for extending the encoding to mixins and traits, and a discussion of related work appears in section 6.

## 2 A review of Links

This section is a summary of the untyped Links representation by Fisher et al. [17]. The syntax of expressions appears in Fig. 1. Apart from the variables ($x$), abstractions ($\lambda x.e$), and applications ($e\ e'$) inherited from the untyped $\lambda$-calculus, there are three new features: tuples $\langle e_1, ..., e_n \rangle$, dictionaries $\{l_1 = e_1, ..., l_n = e_n\}$, and natural numbers $n$.

Tuples are indexed by natural numbers ($e @ i$). They also support functional update and extension. The expression $e @ i \leftarrow e'$ produces a new tuple just like $e$, but with the value at offset $i$ replaced by $e'$. The expression $e \wr \langle e_1, ..., e_n \rangle$ produces a new tuple containing all the values in tuple $e$ followed by the values $e_1$ through $e_n$. Functional update will be used to implement *overriding,* while extension is helpful for *inheritance.*

Dictionaries map *labels* $l$ to values. The expression $e \# l$ fetches the value corresponding to label $l$ in dictionary $e$; this is a more expensive operation than fetching a value from a given offset in a tuple.

For the purpose of representing offsets (or *slots*) within tuples, we need only natural constants and addition. To write real programs, we would need more data types, conditionals, and recursive functions. These features are orthogonal, and omitted from the formal presentation for brevity (although we sometimes use them in examples). The primitive reductions in Fig. 2 may help to elucidate these operations. The original paper [17] includes more details, such as the definition of values ($v$) and evaluation contexts. We will recast these details in a typed setting in section 3.

The most general strategy for encoding objects is this: represent a method suite as a tuple of functions (also known as a virtual function table, or *vtable*), and use a dictionary $d$ to map method labels to natural numbers, representing the corresponding slots in the vtable. Objects are tuples with a pointer to the vtable (shared by all objects created by that class). If the vtable is in the first slot (offset zero) of the object $x$, then the self-application expression for invoking a method named $m$ would be $((x @ 0) @ (d \# m))\ x$.



$$\begin{array}{rcll}
n_1 + n_2 & \rightsquigarrow & n_3 & \text{where } n_3 = n_1 + n_2 \\
(\lambda x.e)\ v & \rightsquigarrow & e[v/x] & \\
\langle v_0, ..., v_{n-1} \rangle @ i & \rightsquigarrow & v_i & \text{where } i < n \\
\langle v_0, ..., v_{n-1} \rangle @ i \leftarrow v' & \rightsquigarrow & \langle v_0, ..., v_{i-1}, v', v_{i+1}, ..., v_{n-1} \rangle & \text{where } i < n \\
\langle v_0, ..., v_{n-1} \rangle \wr \langle v'_0, ..., v'_{m-1} \rangle & \rightsquigarrow & \langle v_0, ..., v_{n-1}, v'_0, ..., v'_{m-1} \rangle & \\
\{l_0 = v_0, ..., l_{n-1} = v_{n-1}\} \# l & \rightsquigarrow & v_i & \text{where } l = l_i
\end{array}$$

**Fig. 2.** Links reduction rules.

```
let CircleBBox = λ⟨sz, vt, dc⟩.
  let center_offset = dc # center in
  let radius_offset = dc # radius in
  let area_offset = dc # area in
  let dc' = { center = center_offset, radius = radius_offset, area = area_offset, bounds = sz } in
  let area_super = vt @ area_offset in
  let area = λself. (area_super self) * 4 / PI in
  let bounds = λself. let ⟨x,y⟩ = ((self @ 0) @ center_offset) self in
                      let r = ((self @ 0) @ radius_offset) self in
                      ⟨⟨x − r, y − r⟩, ⟨x + r, y + r⟩⟩ in
  let vt' = (vt @ area_offset ← area) ≀ ⟨bounds⟩ in ⟨sz + 1, vt', dc'⟩
```

**Fig. 3.** Translation of simple class generator into Links. We take several liberties with the syntax: **let** $x = e$ **in** $e'$ is the obvious syntactic sugar for $((\lambda x.e')\ e)$, but we also permit pattern-matching on tuples.

There is of course an important connection between the dictionary and the vtable in this representation, but they need not be packaged together. To compile a language (such as Moby or OCaml) in which base classes become known at link time, the dictionary would be a module parameter. All dictionary applications would be lifted to the top level of each module, so they occur at link time (i.e., functor application time). To compile Loom, in which classes are first-class values, a dictionary will need to be packaged with each object and passed around at run time. To compile Java, the dictionary is not needed at all, because the layout of the super class vtable is completely known at compile time.[1]

We can represent each class as a triple: the vtable and the dictionary, together with the *size* of the vtable. The size is needed so that when we extend non-manifest base classes, we can compute the offsets of new methods added to the vtable. We omit fields and constructors for convenience, but they pose no additional problems. A class that inherits from an unknown base class is therefore represented as a function that generates a new class triple from an existing one. The function is applied once the base class is provided. Figure 3 shows a rough translation of the example from section 1.

`CircleBBox` is a function whose argument is a triple representing a super class. We begin the function by looking up the offsets of all the methods in the super class,

[1] Here, we assume compilation to native code, which is done dynamically in many implementations. The observation is not true when producing JVM class files, which make extensive use of symbolic references and enjoy binary compatibility.

and then constructing the dictionary for the new class we are generating. It has one new method (bounds), so the new vtable will be larger by one slot. Next, we fetch the existing implementation of area from the super class's vtable vt; it will be called in the new implementation of area. In the implementation of bounds, we invoke two methods on self. We assume that an object is represented as a tuple with a pointer to its vtable at offset zero. In the final **let** expression, we create the new vtable using the functional update and tuple extension operators.

Fisher et al. [17] give further examples and justification for this encoding. Our goal in this paper is to achieve the benefits of Links in a typed representation. There appear to be two relatively independent problems here: (1) develop a sound but flexible type system for the Links primitives, and (2) reflect the various subtype relationships of the source language into the intermediate language.

Both of these problems are hard. In the first case, it is not just a matter of assigning standard types—such as those developed by Cardelli and Mitchell [7]—to dictionary lookup and tuple extension. The way the operators are used in Links, a given dictionary will map method names to offsets in some set of tuples. Although we know nothing about the size or structure of a tuple, we can use it anyway because some dictionary told us where to find the method we need. Subtle invariants govern how these data structures are linked to each other. To type-check Links, we must capture those invariants in the type system.

As for the second problem, Links is intended to be a common intermediate language for various class-based object-oriented languages. Such languages can have wildly different notions of subtyping and subsumption, from the simple name-based class and interface relationships in Java to explicit upward casts in OCaml to the matching relation and match types in Loom [5]. One thing working in our favor at the intermediate language level is that subsumption—where an object of one type may directly be treated as an object of another (super) type—is not strictly necessary. The compiler may insert explicit coercions that adjust the types of objects as needed—with no impact on the run-time behavior—as long as these coercions are proved sound.

# 3 A new typed intermediate language

Shao et al. [30] introduced a framework "for explicitly representing complex propositions and proofs in typed intermediate and assembly languages." The set of types that classify computation terms is defined within the Calculus of Inductive Constructions (CIC) [12]. The semantics of the computation language can then incorporate propositions and proofs expressed in CIC.

As an example, Shao et al. define a language with an unchecked array access operator. One of its operands (apart from the array and the index) is a *proof* that the index is less than the length of the array. If both numbers are known at compile-time, generating these proofs as constants is quite easy. Otherwise, the `if` expression—used to check the index against the bound dynamically—provides proofs to its branches that relate to the semantics of its test expression. This language permits safe bounds check elimination.

The full power of CIC is available in generating the proofs, but they are (like types) compile-time phenomena only: once an expression is shown to be well-formed, the



proofs and types may be erased and have no impact on the behavior and performance of the program.

The Calculus of Constructions [11] rests on the most powerful corner of the $\lambda$ cube [2]. It can encode Church's higher-order predicate logic via the Curry-Howard isomorphism [20]. Extended with inductive definitions, it is the basis for the Coq Proof Assistant [10]. In this paper, we will use a typographically-enhanced variant of Coq 8 syntax.[2] In fact, the definitions in this paper are automatically extracted and sent to Coq for verification.

CIC is most conveniently expressed as a pure type system, where abstractions and applications at different levels are expressed in a uniform syntax, but classified under different *sorts*. The sorts of CIC include SET, PROP, and TYPE. We will use meta-variables $\tau$, $\sigma$, $\kappa$, and $f$ to range over CIC terms, where $\tau$ is usually used for terms corresponding to traditional types, $\kappa$ for terms corresponding to traditional kinds, $f$ for type functions, and $\sigma$ for everything else. The dependent product type is written as $\Pi\alpha{:}\sigma_1.\sigma_2$, or as $\sigma_1 \to \sigma_2$ if $\alpha$ does not appear free in $\sigma_2$. This type is introduced by abstractions of the form $\lambda\alpha{:}\sigma_1.\sigma_2$ and eliminated by applications $\sigma_1\ \sigma_2$. The calculus supports inductive definitions, constructors, and dependent elimination. We freely use the Coq `match` and `Fixpoint` syntax for eliminations, as well as other syntactic niceties like implicit arguments.

## 3.1 Syntax of types and terms

Our first task is to define a set of types for our computation language, LITL.[3] From the Coq library, we import *option* constructor and the definition *nat* : SET of natural numbers in terms of zero (*O*) and the successor function (*S*). We will also need *sym* : SET to represent labels in the dictionary type. Symbols could be represented as natural numbers, or defined (as in appendix D) as sequences of characters from some alphabet. Here is the inductive definition of types in LITL:

**Inductive** $Ty : \textsc{Set} \equiv$
$\quad | \ arw : Ty \to Ty \to Ty$
$\quad | \ snat : nat \to Ty$
$\quad | \ tup : nat \to (nat \to Ty) \to Ty$
$\quad | \ dict : (sym \to option\ Ty) \to Ty$
$\quad | \ mu' : \Pi k : \textsc{Set}.\, (k \to Ty) \to Ty$
$\quad | \ all : \Pi k : \textsc{Set}.\, (k \to Ty) \to Ty$
$\quad | \ ex : \Pi k : \textsc{Set}.\, (k \to Ty) \to Ty.$
**Definition** $mu \equiv mu'\ (k \equiv Ty).$

*arw* $\tau_1\ \tau_2$ is the type of a function mapping values of $\tau_1$ to values of $\tau_2$. *snat* $\widehat{n}$ is the singleton type of the natural number $n$; that is, the value 0 has type *snat O* and the expression $1+1$ has type *snat* $(S\ (S\ O))$. *tup* $\widehat{n}\ f$ is the type of a tuple of size $n$ where $f$ is a type function which maps the index of each field to its type. *dict* $f$ is the

[2] With version 8, Coq moved to a weaker, predicative variant of CIC. We need the impredicative version, which is available with the command-line argument `-impredicative-set`.

[3] LITL Is Typed Links.

type of a dictionary where $f$ is a type function that maps each label to the type of its corresponding value. *mu* $f$, *all* $\kappa$ $f$, and *ex* $\kappa$ $f$ are the *higher-order abstract syntax* encoding [27] of resp. the iso-recursive type $\mu x.f\ x$, the universally quantified type $\forall x{:}\kappa.f\ x$, and the existential type $\exists x{:}\kappa.f\ x$.

To classify an unknown natural number, we hide its value using an existential type:

**Definition** *some_nat* : *Ty* $\equiv$ *ex snat*.

(Thanks to Coq's implicit arguments feature, the $k$ parameter of *ex* is inferred from the type of *snat*.) We can define syntactic sugar for other useful types:

**Definition** *void* : *Ty* $\equiv$ *all* $(\lambda t.\ t)$.
**Definition** *unit* : *Ty* $\equiv$ *ex* $(\lambda t.\ t)$.

The idea is that no values inhabit *void* (more commonly written as $\forall\alpha{:}\mathit{Ty}.\alpha$), and a value of type *unit* has no property.

Tuples are described by their size, and a (type-level) function that maps indices to component types. To specify the function, we will often build a list of types and pass it to the *ith* function:

**Definition** *ith* : *list Ty* $\to$ *nat* $\to$ *Ty* $\equiv$ $\lambda l\ i.$ *nth i l void*.

We are using *list* and *nth* from the Coq library. Lists are constructed from *nil* and *cons* (::), and *nth* has type $\Pi\alpha{:}\textsc{Set}.\mathit{snat} \to \mathit{list}\ \alpha \to \alpha \to \alpha$, where the $\alpha$ is implicit. We use *void* as the default case, for when the index is out of range. Pairs and triples are used fairly often in our encodings, so it is helpful to define more syntactic sugar:

**Definition** $\mathit{tup}_2$ : *Ty* $\to$ *Ty* $\to$ *Ty* $\equiv$ $\lambda t\ u.$ *tup 2* (*ith* $(t :: u :: \mathit{nil})$).
**Definition** $\mathit{tup}_3$ : *Ty* $\to$ *Ty* $\to$ *Ty* $\to$ *Ty* $\equiv$ $\lambda t\ u\ v.$ *tup 3* (*ith* $(t :: u :: v :: \mathit{nil})$).

Dictionaries are described by a (partial) function that maps labels to types. The function relies on the *option* : SET $\to$ SET type constructor of Coq, which is either *None* : $\Pi\alpha{:}$ SET.*option* $\alpha$ or *Some* : $\Pi\alpha{:}\textsc{Set}.\alpha \to \mathit{option}\ \alpha$. Again, we specify the function using a list (in this case a list of pairs, representing a map) and a *lookup* function:

**Definition** *map* : SET $\equiv$ *list* (*prod sym Ty*).
**Fixpoint** *lookup* $(m : \mathit{map})$ $(x : \mathit{sym})$ $\{\mathit{struct}\ m\}$ : *option Ty* $\equiv$
**match** *m* **with** *nil* $\Rightarrow$ *None* | $(y, v) :: m \Rightarrow$ *ifeq x y* (*Some v*) (*lookup m x*) **end**.

The syntax of the type-annotated computation language appears in Fig. 4. It is essentially the same syntax as the untyped version in Fig. 1, but we add a few type operators and annotations.

The tuple selection and update operators now expect a CIC expression $\sigma$, representing a *proof* that the index is less than the size of the tuple. (We use $\mathit{lt} : \mathit{nat} \to \mathit{nat} \to$ PROP from the Coq library.) The labels in the dictionary construction and lookup syntax are CIC expressions of set *sym*. We also added standard type manipulation terms such as the type abstraction $\Lambda\alpha{:}\sigma.f$ and its corresponding type instantiation $e\,[\tau]$, existential package constructor $[\tau_1, e \triangleright \tau_2]$ and its corresponding destructor (**open** $e_1$ **as** $[\alpha, x]$ **in** $e_2$), as well as recursive type folding (**fold** $e$ **as** $\tau$) and unfolding (**unfold** $e$). Finally, there is a cast expression (**cast** $[\sigma]\,e$). Here, $\sigma$ should be a proof that $\mathit{eq}\ \tau_1\ \tau_2$. Then, if $e$ has type $\tau_1$, the entire cast expression can be considered to have type $\tau_2$. See the typing rules in section 3.3.



$$
\begin{array}{l}
e ::= x \mid n \mid e_1 + e_2 \mid f \mid e_1\ e_2 \mid e\,[\tau] \mid \langle e_1, \ldots, e_n \rangle \mid e_1\,@\,e_2\,[\sigma] \mid e_1\,@\,e_2\,[\sigma] \leftarrow e_3 \\
\quad \mid e \fatsemi \langle e_1, \ldots, e_n \rangle \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e \# l\,[\sigma] \mid \textsf{cast}\,[\sigma]\,e \mid [\tau_1, e \triangleright \tau_2] \\
\quad \mid \textsf{open}\ e_1\ \textsf{as}\ [\alpha, x]\ \textsf{in}\ e_2 \mid \textsf{fold}\,e\ \textsf{as}\ \tau \mid \textsf{unfold}\,e \\
f ::= \lambda x{:}\tau.e \mid \Lambda\alpha{:}\sigma.f
\end{array}
$$

**Fig. 4.** LITL term syntax.

### 3.2 Dynamic semantics

The dynamic semantics of LITL are defined in terms of a small-step reduction $\rightsquigarrow$. We distinguish a subset of the expressions as *values*. The primitive reduction rules are the only enlightening part; the definition of values and congruence rules are available in appendix A.

**Primitive reductions** $\boxed{e \rightsquigarrow e'}$

$$\frac{}{n_1 + n_2 \rightsquigarrow n_3 \quad \text{where } n_3 = n_1 + n_2}\ (1) \qquad \frac{}{(\lambda x{:}\_.e)\ v \rightsquigarrow e[v/x]}\ (2)$$

$$\frac{}{(\Lambda\alpha{:}\_.f)\,[\tau] \rightsquigarrow f[\tau/\alpha]}\ (3) \qquad \frac{}{\textsf{cast}\,[\_]\,v \rightsquigarrow v}\ (4)$$

$$\frac{}{\textsf{open}\ [\tau, v \triangleright \_]\ \textsf{as}\ [\alpha, x]\ \textsf{in}\ e \rightsquigarrow e[v/x][\tau/\alpha]}\ (5) \qquad \frac{}{\textsf{unfold}\,(\textsf{fold}\,v\ \textsf{as}\ \tau) \rightsquigarrow v}\ (6)$$

$$\frac{}{\langle v_1, \ldots, v_n \rangle\,@\,i\,[\_] \leftarrow v' \rightsquigarrow \langle v_1, \ldots, v_i, v', v_{i+2}, \ldots, v_n \rangle}\ (7)$$

$$\frac{}{\langle v_1, \ldots, v_n \rangle \fatsemi \langle v'_1, \ldots, v'_m \rangle \rightsquigarrow \langle v_1, \ldots, v_n, v'_1, \ldots, v'_m \rangle}\ (8) \qquad \frac{}{\langle v_1, \ldots, v_n \rangle\,@\,i\,[\_] \rightsquigarrow v_{i+1}}\ (9)$$

$$\frac{}{\{l_1 = v_1, \ldots, l_n = v_n\} \# l_i\,[\_] \rightsquigarrow v_i}\ (10)$$

### 3.3 Static semantics

To specify the static semantics of this language, one more definition will be needed:

**Fixpoint** *append* $(n : \mathit{nat})$ $(f\ g : \mathit{nat} \to \mathit{Ty})$ $(i : \mathit{nat})$ $\{\ \mathit{struct}\ i\ \}$ : *Ty* $\equiv$
**match** *i* **with** *O* $\Rightarrow$ (**match** *n* **with** *O* $\Rightarrow$ *g O* | _ $\Rightarrow$ *f O* **end**)
| *S i* $\Rightarrow$ **match** *n* **with** *O* $\Rightarrow$ *g* (*S i*)
| *S n* $\Rightarrow$ *append n* $(\lambda x.f\ (S\ x))$ *g i*
**end**
**end**.

The judgments are $\Delta \vdash^{\text{CIC}} \tau : \sigma$ from the type language and $\Delta; \Gamma \vdash e : \tau$ for term formation. The environment $\Delta$ maps type variables to their kinds, while $\Gamma$ maps term variables to their types. LITL enjoys the subject reduction and progress properties; proofs are in appendix B.

**Term formation** $\boxed{\Delta;\Gamma \vdash e : \tau}$

$$\frac{\Delta \vdash^{\text{CIC}} \Gamma(x) : Ty}{\Delta;\Gamma \vdash x : \Gamma(x)} \quad (11) \qquad \frac{}{\Delta;\Gamma \vdash n : snat\ \widehat{n}} \quad (12)$$

$$\frac{\Delta;\Gamma \vdash e_1 : snat\ \tau_1 \quad \Delta;\Gamma \vdash e_2 : snat\ \tau_2}{\Delta;\Gamma \vdash e_1 + e_2 : snat\ (plus\ \tau_1\ \tau_2)} \quad (13) \qquad \frac{\Delta \vdash^{\text{CIC}} \tau : Ty \quad \Delta;\Gamma, x{:}\tau \vdash e : \tau'}{\Delta;\Gamma \vdash \lambda x{:}\tau.e : arw\ \tau\ \tau'} \quad (14)$$

$$\frac{\Delta \vdash^{\text{CIC}} \sigma : \text{SET} \qquad \Delta, \alpha{:}\sigma;\Gamma \vdash f : \tau \qquad \alpha \notin \Delta}{\Delta;\Gamma \vdash \Lambda\alpha{:}\sigma.f : all\ (\lambda\alpha{:}\sigma.\tau)} \quad (15) \qquad \frac{\Delta;\Gamma \vdash e_1 : arw\ \tau'\ \tau \quad \Delta;\Gamma \vdash e_2 : \tau'}{\Delta;\Gamma \vdash e_1\ e_2 : \tau} \quad (16)$$

$$\frac{\Delta;\Gamma \vdash e : all\ \tau' \qquad \Delta \vdash^{\text{CIC}} \tau : \sigma'}{\Delta;\Gamma \vdash e\,[\tau] : \tau'\ \tau} \quad (17) \qquad \frac{\Delta \vdash^{\text{CIC}} \sigma : eq\ \tau_1\ \tau_2 \quad \Delta;\Gamma \vdash e : \tau_1}{\Delta;\Gamma \vdash \mathbf{cast}\,[\sigma]\,e : \tau_2} \quad (18)$$

$$\frac{\Delta \vdash^{\text{CIC}} \tau_1 : \sigma \quad \Delta \vdash^{\text{CIC}} \tau_2 : \sigma \to Ty \quad \Delta \vdash^{\text{CIC}} \sigma : \text{SET} \quad \Delta;\Gamma \vdash e : \tau_2\ \tau_1}{\Delta;\Gamma \vdash [\tau_1, e \triangleright \tau_2] : ex\ \tau_2} \quad (19) \qquad \frac{\Delta;\Gamma \vdash e : ex\ \tau \qquad \Delta \vdash^{\text{CIC}} \tau' : Ty \quad \Delta, \alpha{:}\sigma;\Gamma, x{:}(\tau\ \alpha) \vdash e' : \tau'}{\Delta;\Gamma \vdash \mathbf{open}\ e\ \mathbf{as}\ [\alpha, x]\ \mathbf{in}\ e' : \tau'} \quad (20)$$

$$\frac{\Delta;\Gamma \vdash e : \tau\ (mu\ \tau)}{\Delta;\Gamma \vdash \mathbf{fold}\,e\,\mathbf{as}\,\tau : mu\ \tau} \quad (21) \qquad \frac{\Delta;\Gamma \vdash e : mu\ \tau}{\Delta;\Gamma \vdash \mathbf{unfold}\,e : \tau\ (mu\ \tau)} \quad (22)$$

$$\frac{\Delta;\Gamma \vdash e_i : \tau\ \widehat{i} \quad \forall i < n}{\Delta;\Gamma \vdash \langle e_0, \ldots, e_{n-1} \rangle : tup\ \widehat{n}\ \tau} \quad (23)$$

$$\frac{\Delta;\Gamma \vdash e_1 : tup\ \sigma_1\ \tau_1 \qquad \Delta;\Gamma \vdash e_2 : snat\ \sigma_2 \qquad \Delta \vdash^{\text{CIC}} \sigma : lt\ \sigma_2\ \sigma_1}{\Delta;\Gamma \vdash e_1 \mathbin{@} e_2\,[\sigma] : \tau_1\ \sigma_2} \quad (24)$$

$$\frac{\Delta;\Gamma \vdash e_1 : tup\ \sigma_1\ \tau_1 \qquad \Delta;\Gamma \vdash e_2 : snat\ \sigma_2 \quad \Delta;\Gamma \vdash e_3 : \tau_1\ \sigma_2 \qquad \Delta \vdash^{\text{CIC}} \sigma : lt\ \sigma_2\ \sigma_1}{\Delta;\Gamma \vdash e_1 \mathbin{@} e_2\,[\sigma] \leftarrow e_3 : tup\ \sigma_1\ \tau_1} \quad (25)$$

$$\frac{\Delta;\Gamma \vdash e : tup\ \tau_1\ \tau_2 \qquad \Delta;\Gamma \vdash \langle e_1, \ldots, e_n \rangle : tup\ \tau_1'\ \tau_2'}{\Delta;\Gamma \vdash e \mathbin{\S} \langle e_1, \ldots, e_n \rangle : tup\ (plus\ \tau_1\ \tau_1')\ (append\ \tau_1\ \tau_2\ \tau_2')} \quad (26)$$

$$\frac{\Delta;\Gamma \vdash e_i : \tau_i \quad \wedge \quad \tau\ \widehat{l_i} = Some\ \tau_i \qquad \forall i < n \quad l \notin l \Rightarrow \tau\ \widehat{l} = None}{\Delta;\Gamma \vdash \{l_0 = e_0, \ldots, l_{n-1} = e_{n-1}\} : dict\ \tau} \quad (27)$$

$$\frac{\Delta;\Gamma \vdash e : dict\ \tau \quad \Delta \vdash^{\text{CIC}} \sigma : eq\ (\tau\ \widehat{l})\ (Some\ \tau')}{\Delta;\Gamma \vdash e \# l\,[\sigma] : \tau'} \quad (28) \qquad \frac{\Delta;\Gamma \vdash e : \tau \qquad \tau =_{\beta\eta\iota} \tau'}{\Delta;\Gamma \vdash e : \tau'} \quad (29)$$



# 4 Typed compilation of classes

We now return to the running example, whose Links translation was provided in figure 3. In this section, we will develop the typed encoding of that example in stages, showing additionally how objects are created from classes, and how various implementations of the base class `circle` can be specified.

## 4.1 Class representation

Recall that in Links, *CircleBBox* was represented as a function that generates a new class from a given one. The class argument was depicted as a triple $\langle \mathsf{sz}, \mathsf{vt}, \mathsf{dc} \rangle$. We know very little about this (non-manifest) base class: the size and layout of the vtable (vt) are unknown. We just know that the dictionary (dc) contains bindings for the three known methods: center, radius, and area. Moreover, the dictionary maps the method names to offsets that may be applied to the vt to select functions of the correct type. Many different representations of this base class are possible.

The components of the class triple must be typed, so we begin by supposing that sz has type *snat n* (for some *n*), that vt has type *tup n f* (for some *f*), and finally that dc has type *dict g* (for some *g*). These three parameters (*n*, *f*, and *g*) uniquely specify the *representation* of a class:

**Definition** *Rep* : SET $\equiv$ $(nat \times (Ty \to nat \to Ty) \times (sym \to option\ Ty))$.
**Definition** *size* $\equiv$ $\lambda r : Rep.$ **match** *r* **with** $(n, \_, \_) \Rightarrow n$ **end**.
**Definition** *tupfn* $\equiv$ $\lambda r : Rep.$ **match** *r* **with** $(\_, f, \_) \Rightarrow f$ **end**.
**Definition** *dictfn* $\equiv$ $\lambda r : Rep.$ **match** *r* **with** $(\_, \_, g) \Rightarrow g$ **end**.

We have made one small departure from the description above: the type of the tuple function *f* includes an extra *Ty* argument. This is because the elements of the tuple are methods, or functions over an explicit self parameter. The *Ty* argument is the type of self. This cannot be fixed in one place, but must be a parameter because the method will be reused in derived classes with different types for self. We will demonstrate how this works in section 4.3.

Let us specify two distinct representations of `circle`, the base class in our example. The methods use floating-point types, which we have not defined formally, but we can suppose that they exist:

**Parameter** *float* : *Ty*.
**Definition** *fpoint* : *Ty* $\equiv$ $tup_2\ float\ float$.
**Definition** *frect* : *Ty* $\equiv$ $tup_2\ fpoint\ fpoint$.

Additionally, *fpoint* is a pair of floats, and *frect* is a pair of points (for the bounds method). Here is the simplest representation, where the three methods appear in order in the vtable, with nothing extra:

**Definition** *circA_rep* : *Rep* $\equiv$
$(3,\ \lambda self.\ ith\ (arw\ self\ fpoint :: arw\ self\ float :: arw\ self\ float :: nil),$
$lookup\ ((center,\ snat\ 0) :: (radius,\ snat\ 1) :: (area,\ snat\ 2) :: nil))$.

With this representation, we have the following equivalences in CIC:

$$\begin{aligned} \mathit{size}\ \mathit{circA\_rep} &=_{\beta\eta\iota} 3 \\ \mathit{dictfn}\ \mathit{circA\_rep}\ \mathit{center} &=_{\beta\eta\iota} 0 \\ \mathit{tupfn}\ \mathit{circA\_rep}\ \tau\ 0 &=_{\beta\eta\iota} \mathit{arw}\ \tau\ \mathit{fpoint} \end{aligned}$$

We can encode a more complex representation, where the methods appear in different slots, and some slots are taken up by unknown values:

**Definition** $\mathit{circB\_rep} : \mathit{Rep} \equiv$
$(5, \lambda \mathit{self}.\ \mathit{ith}\ (\mathit{arw}\ \mathit{self}\ (\mathit{ex}\ \mathit{snat}) :: \mathit{arw}\ \mathit{self}\ \mathit{float} :: \mathit{arw}\ \mathit{self}\ \mathit{fpoint} ::$
$\mathit{snat}\ 0 :: \mathit{arw}\ \mathit{self}\ \mathit{float} :: \mathit{nil}),$
$\mathit{lookup}\ ((\mathit{radius}, \mathit{snat}\ 4) :: (\mathit{area}, \mathit{snat}\ 1) :: (\mathit{center}, \mathit{snat}\ 2) :: \mathit{nil})).$

Here, slots 0 and 3 are taken up by other values; one of them is not even a function. Still, the *dictfn* tells us where to find the three circle methods.

### 4.2 Class specification

Now, how do we ensure that the three *Rep* components $(n, f, g)$ correspond with one another? The constraint, roughly, is that for each method $m$, there exists some $j : \mathit{nat}$ such that $j < n$ and $g\ m = \mathit{Some}\ (\mathit{snat}\ j)$ and $f\ j = \tau$ where $\tau$ is the expected type of the method. We can encode precisely this property in CIC:

**Inductive** $\mathit{HasMethod}\ (r : \mathit{Rep})\ (m : \mathit{sym})\ (t : \mathit{Ty}) : \textsc{Set} \equiv$
$\mathit{method} : \Pi i : \mathit{nat}.\, \mathit{lt}\ i\ (\mathit{size}\ r) \to \mathit{eq}\ (\mathit{dictfn}\ r\ m)\ (\mathit{Some}\ (\mathit{snat}\ i)) \to$
$(\Pi \mathit{self}.\, \mathit{eq}\ (\mathit{tupfn}\ r\ \mathit{self}\ i)\ (\mathit{arw}\ \mathit{self}\ t)) \to \mathit{HasMethod}\ r\ m\ t.$

Notice that the offset $i$ is specified in the *method* constructor, but does not appear in the *HasMethod* term itself. This is a form of *dependent pair,* and thanks to the dependent elimination feature of CIC, we can create selectors that mimic the *dot notation* described by Cardelli and Leroy [6]. Here is the term to fetch the offset:

**Definition** $\mathit{offset} \equiv \lambda r\ m\ t.\ \lambda p : \mathit{HasMethod}\ r\ m\ t.$
**match** $p$ **with** $\mathit{method}\ i\ \mathit{pf}\ \mathit{dc}\ \mathit{tp} \Rightarrow i$ **end**.

The other selectors have return types that include the *offset* of the parameter itself.

**Definition** $\mathit{proof} \equiv \lambda r\ m\ t.\ \lambda p : \mathit{HasMethod}\ r\ m\ t.$
**match** $p$ **as** $q$ **return** $\mathit{lt}\ (\mathit{offset}\ q)\ (\mathit{size}\ r)$ **with** $\mathit{method}\ i\ \mathit{pf}\ \mathit{dc}\ \mathit{tp} \Rightarrow \mathit{pf}$ **end**.
**Definition** $\mathit{dicteq} \equiv \lambda r\ m\ t.\ \lambda p : \mathit{HasMethod}\ r\ m\ t.$
**match** $p$ **as** $q$ **return** $\mathit{eq}\ (\mathit{dictfn}\ r\ m)\ (\mathit{Some}\ (\mathit{snat}\ (\mathit{offset}\ q)))$
**with** $\mathit{method}\ i\ \mathit{pf}\ \mathit{dc}\ \mathit{tp} \Rightarrow \mathit{dc}$ **end**.
**Definition** $\mathit{tupeq} \equiv \lambda r\ m\ t.\ \lambda p : \mathit{HasMethod}\ r\ m\ t.$
**match** $p$ **as** $q$ **return** $\Pi s.\, \mathit{eq}\ (\mathit{tupfn}\ r\ s\ (\mathit{offset}\ q))\ (\mathit{arw}\ s\ t)$
**with** $\mathit{method}\ i\ \mathit{pf}\ \mathit{dc}\ \mathit{tp} \Rightarrow \mathit{tp}$ **end**.

So, if we had some evidence that a representation $r$ has a method *center* returning an *fpoint*, it would be expressed as a term $p : \mathit{HasMethod}\ r\ m\ \mathit{fpoint}$. We can tuple several *HasMethod* terms to create a *signature* for a class:



**Definition** $\mathit{circ\_signature} \equiv \lambda r.$
$(\mathit{HasMethod}\ r\ \mathit{center}\ \mathit{fpoint} \times \mathit{HasMethod}\ r\ \mathit{radius}\ \mathit{float} \times$
$\mathit{HasMethod}\ r\ \mathit{area}\ \mathit{float}).$

Now we create a term to use as evidence that *circB_rep* meets the *circ_signature*. It consists of proofs that the indices in the dictionary are less than the tuple size, that the types in the vtable match the signature, and so on.

**Definition** $\mathit{self\_equal} \equiv \lambda t\ s.\ \mathit{refl\_equal}\ (\mathit{arw}\ s\ t).$
**Definition** $\mathit{circB\_witness} : \mathit{circ\_signature}\ \mathit{circB\_rep} \equiv$
$(\mathit{method}\ \mathit{circB\_rep}\ \mathit{center}\ (\mathit{le\_S}\ (\mathit{le\_S}\ (\mathit{le\_n}\ 3)))\ (\mathit{refl\_equal}\ \_)\ (\mathit{self\_equal}\ \mathit{fpoint}),$
$\mathit{method}\ \mathit{circB\_rep}\ \mathit{radius}\ (\mathit{le\_n}\ 5)\ (\mathit{refl\_equal}\ \_)\ (\mathit{self\_equal}\ \mathit{float}),$
$\mathit{method}\ \mathit{circB\_rep}\ \mathit{area}\ (\mathit{le\_S}\ (\mathit{le\_S}\ (\mathit{le\_S}\ (\mathit{le\_n}\ 2))))\ (\mathit{refl\_equal}\ \_)\ (\mathit{self\_equal}\ \mathit{float})).$

Not all of the *method* parameters need to be specified, thanks to Coq's implicit arguments feature. The offset of each method, for example, is inferred from the proof term. The center method appears at offset 2, so we must show that $2 < 5$. The *lt* relation in the Coq library is specified in terms of *le* (less than or equal): $\mathit{lt}\ i\ n \equiv \mathit{le}\ (S\ i)\ n$. The term $\mathit{le\_n}\ 3$ is the proof of $3 \leq 3$, and the two *le_S* constructors transform that into a proof of $3 \leq 5$ or, equivalently, $2 < 5$. We define projections over *circ_signature* types, to be used later in examples:

**Definition** $\mathit{circ\_center} : \Pi r.\, \mathit{circ\_signature}\ r \to \mathit{HasMethod}\ r\ \mathit{center}\ \mathit{fpoint} \equiv$
$\lambda r\ p.$ **match** $p$ **with** $(\mathit{ce}, \mathit{ra}, \mathit{ar}) \Rightarrow \mathit{ce}$ **end**.
**Definition** $\mathit{circ\_radius} : \Pi r.\, \mathit{circ\_signature}\ r \to \mathit{HasMethod}\ r\ \mathit{radius}\ \mathit{float} \equiv$
$\lambda r\ p.$ **match** $p$ **with** $(\mathit{ce}, \mathit{ra}, \mathit{ar}) \Rightarrow \mathit{ra}$ **end**.
**Definition** $\mathit{circ\_area} : \Pi r.\, \mathit{circ\_signature}\ r \to \mathit{HasMethod}\ r\ \mathit{area}\ \mathit{float} \equiv$
$\lambda r\ p.$ **match** $p$ **with** $(\mathit{ce}, \mathit{ra}, \mathit{ar}) \Rightarrow \mathit{ar}$ **end**.

### 4.3 Object types and method invocation

Now that we can encode class representations (and constraints on them), we are ready to define the types of objects. In this section, we will represent an object as a pair containing the dictionary and the vtable. We ignore object fields throughout this work, because they are orthogonal. Also, we mentioned before that in Moby and OCaml, where classes can be functor parameters, it is not necessary to package the dictionary with each object. In section 5, we demonstrate an optimized encoding that separates the two components, so that dictionary lookups can be hoisted to the module level. Here is the type of an object pair, given a class representation and the type of self:

**Definition** $\mathit{objrep} : \mathit{Rep} \to \mathit{Ty} \to \mathit{Ty} \equiv \lambda r\ \mathit{self}.$
$\mathit{tup}_2\ (\mathit{dict}\ (\mathit{dictfn}\ r))\ (\mathit{tup}\ (\mathit{size}\ r)\ (\mathit{tupfn}\ r\ \mathit{self})).$

The self type is resolved with a fixpoint, indicating that the self parameter must be an object of exactly the same type as the object containing the method.

**Definition** $\mathit{selfty} : \mathit{Rep} \to \mathit{Ty} \equiv \lambda r.\ \mathit{mu}\ (\mathit{objrep}\ r).$

```
let invoke_radius = λx:objty circ_signature.
  open x as [r, x1] in open x1 as [p, x2] in let x3 = unfold x2 in
  let dc = x3 @0 [lt02] in let vt = x3 @1 [lt12] in
  let j = dc # radius [dicteq (circ_radius p)] in
  let f = vt @j [proof (circ_radius p)] in
  let f = cast [tupeq (circ_radius p) (selfty r)] f in f x2
```

**Fig. 5.** Code to invoke the radius method on an object $x$.

Finally, we must hide the representation type. Two existential quantifiers are used here. The outer one hides the *Rep*, while the inner one hides the evidence that the representation matches some specified signature.

**Definition** $objty'' : \Pi sig : Rep \to \text{SET}.\, \Pi r.\, sig\ r \to Ty \equiv \lambda sig\ r\ \_.\ selfty\ r.$
**Definition** $objty' : (Rep \to \text{SET}) \to Rep \to Ty \equiv \lambda sig\ r.\ ex\ (objty''\ sig\ r).$
**Definition** $objty : (Rep \to \text{SET}) \to Ty \equiv \lambda sig.\ ex\ (objty'\ sig).$

So, the type of a circle object is *objty circ_signature*. In more conventional notation, the object encoding is:

$$\exists r : Rep.\, \exists p : circ\_signature\ r.\, \mu\alpha : Ty.\, objrep\ r\ \alpha$$

(It is not necessary to split the existentials over three Coq definitions, but it allows for shorter annotations in some programs.)

Now we present a function that invokes the radius method on an object $x$. In section 2, with untyped terms, this was written simply as $((x\,@\,1)\,@\,((x\,@\,0)\,\#\,\text{radius}))\ x$. Figure 5 contains a function that takes $x$ as a parameter, and calls radius. The code is shown in A-normal form [18] for readability, but this is not essential. Apart from the open-open-unfold sequence in the beginning, the burden imposed by the type system includes the proof annotations on tuple selection and dictionary lookup, and the cast expression just before the (virtual) function call. The terms *lt02* and *lt12* in the select statements refer to these proof constants:

**Definition** $lt02 : lt\ 0\ 2 \equiv le\_S\ (le\_n\ 1).$
**Definition** $lt12 : lt\ 1\ 2 \equiv le\_n\ 2.$

If the objects contained fields, then these proofs would depend on the number of fields in the tuple. To support this, the existential would also need to hide the size of the tuple, $m$, and a proof of *lt 1 m* (from which the proof of *lt 0 m* could be derived).

These type operators and proof annotations buy quite a lot in terms of flexibility and safety. In languages that support non-manifest base classes, the representations of classes and objects have complex invariants that are now enforced by the type system of the intermediate language.

### 4.4 Class types and instantiation

The type of a class is slightly more complex because the vtable in the class plays a different role than the vtable embedded in an object (even though they are the same data



```
let new_circ = λc0:classy circ_signature.
  open c0 as [r, c1] in open c1 as [p, c2] in
  let dc = c2 @1 [lt13] in let ms = c2 @2 [lt23] in
  let vt = ms [r] [p] in let x = fold ⟨dc, vt⟩ as objrep r in
  [r, [p, x ▷ objty'' circ_signature r] ▷ objty' circ_signature]
```

**Fig. 6.** Create a new circle object, given a circle class.

structure at run time). Methods must be inheritable. This means that the *self* parameter will have different types at different points in the hierarchy. Therefore, in the class, the vtable must be parameterized by the type of self. The only restriction is that self must have at least the methods defined in the class in which the method is defined. We call this parameterized vtable a *method suite:*

**Definition** $methsuite'' : \Pi sig : Rep \to \text{SET}.\, Rep \to \Pi r' : Rep.\, sig\ r' \to Ty \equiv$
$\lambda sig\ r\ r'\ \_.\ tup\ (size\ r)\ (tupfn\ r\ (selfty\ r')).$
**Definition** $methsuite' : (Rep \to \text{SET}) \to Rep \to Rep \to Ty \equiv$
$\lambda sig\ r\ r'.\ all\ (methsuite''\ sig\ r\ r').$
**Definition** $methsuite : (Rep \to \text{SET}) \to Rep \to Ty \equiv$
$\lambda sig\ r.\ all\ (methsuite'\ sig\ r).$

Notice the subtle difference in usage between the representations $r$ and $r'$. The former is the representation of the current class (and determines the methods that appear in the tuple), while the latter is the representation of some subclass that is inheriting these methods. Its only impact is on the type of the self parameter.

We noted previously that each class is represented as a triple. Here is the definition of the triple, in terms of the class signature *sig* and representation $r$.

**Definition** $classtup : (Rep \to \text{SET}) \to Rep \to Ty \equiv$
$\lambda sig\ r.\ tup_3\ (snat\ (size\ r))\ (dict\ (dictfn\ r))\ (methsuite\ sig\ r).$

As with object types, we must conceal the representation along with the proof that it meets the specified signature.

**Definition** $classty'' : \Pi sig : Rep \to \text{SET}.\, \Pi r.\, sig\ r \to Ty \equiv \lambda sig\ r\ \_.\ classtup\ sig\ r.$
**Definition** $classty' : (Rep \to \text{SET}) \to Rep \to Ty \equiv \lambda sig\ r.\ ex\ (classty''\ sig\ r).$
**Definition** $classty : (Rep \to \text{SET}) \to Ty \equiv \lambda sig.\ ex\ (classty'\ sig).$

This way, both the 'A' and 'B' implementations of the circle class can appear to have the same type: *classty circ_signature*.

Figure 6 contains an implementation of the 'new' operator, that creates a new object from a class. It instantiates the method suite with the representation of the provided class, so that the methods will accept the new object as the self argument. Then, the dictionary and vtable are paired together, folded, and re-packaged. As before, *lt13* and *lt23* stand for constant proof terms.

### 4.5 Class declarations

These sophisticated representations of class and object types would be for naught if we are unable to implement a circle class in the first place. In this section, we demonstrate

```
let circB =
  let dc = {radius=4, area=1, center=2} in
  let ms = Λr:Rep.Λp:circ_signature r.
           ⟨λs:selfty r. /* code of type ex snat */, λs:selfty r. /* code of type float */,
            λs:selfty r. /* code of type fpoint */, 0, λs:selfty r. /* code of type float */ ⟩ in
  let c = ⟨5, dc, ms⟩ in
  [circB_rep, [circB_witness, c ▷ classty'' circ_signature circB_rep] ▷ classty' circ_signature]
```

**Fig. 7.** An implementation of the circle class signature.

that the type *classty circ_signature* is habitable: see the definition of the 'B' circle class in figure 7. We do not provide complete implementations of the methods: for that, we would need to define floating-point operations and fields.

With this class, we can now connect together the code in the two previous figures like this: *invoke_radius* (*new_circ circB*). This creates a new circle from *circB*, invokes the radius method of that object, and returns a *float*. We leave it as an exercise to define a different implementation *circA*, using the *circA_rep* defined on page 10.

### 4.6 Extending an unknown base class

Now we have come to the heart of the whole problem: typed compilation against a non-manifest base class. Our running example extends some unknown class (that matches the circle signature) by overriding area and adding a new method bounds. In CIC, we can define a signature for this derived class, *bbox*:

**Definition** *bbox_signature* $\equiv \lambda r.$
$(HasMethod\ r\ center\ fpoint \times HasMethod\ r\ radius\ float \times$
$HasMethod\ r\ area\ float \times HasMethod\ r\ bounds\ frect)$.

The representation of the derived class will of course depend on the layout of its parent. Still, we can define a function to produce a *bbox* representation, given another representation *r* that matches the *circ_signature*:

**Definition** *bbox_rep* : $\Pi r : Rep.\ circ\_signature\ r \to Rep \equiv \lambda r\ p.$
$(plus\ 1\ (size\ r),\ \lambda self.\ append\ (size\ r)\ (tupfn\ r\ self)\ (ith\ (arw\ self\ frect :: nil)),$
$lookup\ ((center,\ snat\ (offset\ (circ\_center\ p))) ::$
$(radius,\ snat\ (offset\ (circ\_radius\ p))) :: (area,\ snat\ (offset\ (circ\_area\ p))) ::$
$(bounds,\ snat\ (size\ r)) :: nil))$.

This works by retrieving the offsets of the inherited methods from the witness *p*, and placing the bounds method in slot *n*—the size of the parent representation. The tuple function uses *append* to join the type of the new method with the types of the parent. With this (parameterized) representation, we have the following:

$$size\ (bbox\_rep\ circB\_witness) =_{\beta\eta\iota} 6$$
$$dictfn\ (bbox\_rep\ circB\_witness)\ \mathsf{center} =_{\beta\eta\iota} Some\ (snat\ 2)$$
$$dictfn\ (bbox\_rep\ circB\_witness)\ \mathsf{bounds} =_{\beta\eta\iota} Some\ (snat\ 5)$$
$$tupfn\ (bbox\_rep\ circB\_witness)\ \tau\ 5 =_{\beta\eta\iota} arw\ \tau\ frect$$



The next step is to prove that the extended representation matches the *bbox* signature. This is more difficult than it may seem at first. It depends critically on the semantics of *append*. First, extending a tuple with new elements does not alter the types of existing elements. Second, the new elements can be retrieved by adding the size of the original tuple to their offsets. These properties are expressed by the following Coq lemmas:

**Lemma** $append\_semantics_1 : \Pi i\ n.\ lt\ i\ n \to \Pi f\ g.\ eq\ (append\ n\ f\ g\ i)\ (f\ i)$.
**Lemma** $append\_semantics_2 : \Pi k\ n\ f\ g.\ eq\ (append\ n\ f\ g\ (plus\ k\ n))\ (g\ k)$.

With these properties, we can prove the following term:

**Definition** *bbox_witness* : $\Pi r.\ \Pi p : circ\_signature\ r.\ bbox\_signature\ (bbox\_rep\ p)$.

As needed, this shows that the extended representation matches the *bbox* signature. (To conserve space, proofs for these properties have been moved to appendix C.)

Just one more definition is needed to extend a non-manifest base class. We instantiate the super class dictionary with the representation of the derived class. This is what permits us to pass *bbox* objects to those *circle* methods. To do this, we must prove that the derived representation still matches the super class signature. Fortunately, this is trivial: just a repackaging of the *HasMethod* properties, to drop the one referring to the bounds method:

**Definition** *bbox2circ* : $\Pi r.\ bbox\_signature\ r \to circ\_signature\ r \equiv$
$\lambda r\ p.$ **match** $p$ **with** $(ce, ra, ar, bo) \Rightarrow (ce, ra, ar)$ **end**.

Figure 8 contains the complete code for extending an unknown base class. It corresponds to the OCaml functor given in the introduction, and is a typed version of the Links code in section 2. Most of the non-trivial typing aspects have already been explained. Look for occurrences of *bbox_rep*, *bbox_witness*, and *bbox2circ* in the typing annotations. In our example, the area method included a super call. We omitted the call itself in the figure (along with the rest of the method bodies), but it works very simply. At the point where we define area_m′, we have already selected the area method from vt, the super class vtable. Within the body of area_m′, we would apply area_m to s to call the super-class method.

Also, notice the **cast** applied to the overridden area method before updating the vtable. It is the inverse of the cast used when selecting a method from the vtable. We just defined area_m′, so it has an arrow type to begin with. But the designated slot of the vtable has an opaque type, literally *tupfn* r (*selfty* r″) (*offset* (*circ_area* p)), which cannot be reduced because r is a variable. But we can use (a symmetric version of) the *tupeq* property to cast from the concrete to the opaque, and then update that slot of the vtable.

## 5 Extensions

This section explores ways to extend the basic techniques in several directions, giving some idea of the versatility of LITL.

```
let circle_bbox = λc:classty circ_signature.
  open c as [r, c] in open c as [p, c] in
  let sz = c@0[lt03] in let dc = c@1[lt13] in let ms = c@2[lt23] in
  let ci = dc#center[dicteq (circ_center p)] in
  let ri = dc#radius[dicteq (circ_radius p)] in
  let ai = dc#area[dicteq (circ_area p)] in
  let dc′ = {center=ci, radius=ri, area=ai, bounds=sz} in
  let ms′ = Λr″:Rep.Λp″:bbox_signature r″.
    let vt = ms[r″][bbox2circ p″] in
    let bounds_m = λs:selfty r″. /* code of type frect */ in
    let area_m = vt@ai[proof (circ_area p)] in
    let area_m = cast[tupeq (circ_area p) (selfty r″)] area_m in
    let area_m′ = λs:selfty r″. /* code of type float */ in
    let area_m′ = cast[sym_eq (tupeq (circ_area p) (selfty r″))] area_m′ in
    let vt′ = vt@ai[proof (circ_area p)] ← area_m′ in
    vt′§⟨bounds_m⟩ in
  let c′ = ⟨1+sz, dc′, ms′⟩ in
  let c′ = [bbox_witness p, c′ ▷ classty″ bbox_signature (bbox_rep p)] in
  [bbox_rep p, c′ ▷ classty′ bbox_signature]
```

**Fig. 8.** Code to extend a non-manifest base class.

```
let upcast = λx:objty bbox_signature.
  open x as [r, x] in open x as [p, x] in
  [r, [bbox2circ p, x ▷ objty″ circ_signature r] ▷ objty′ circ_signature]
```

**Fig. 9.** To upcast a bbox to a circle, we open and repackage the object.

### 5.1 Encoding subsumption as type coercions

Object-oriented languages enjoy *subsumption:* a context expecting an object of type $t$ will be satisfied with an object of some *subtype* of $t$. The precise rules about what constitutes a subtype, and where subsumption may be used, differ with each language.

Our intermediate language does not directly support subtyping. Nevertheless, if we examine object types of two classes in a subclass relationship, we notice they differ only in what is known about the (hidden) representation. It is always possible to open and repackage the object with *less* information about its representation. The example in Fig. 9 casts a bbox object to a circle (its super class). This is done entirely with type coercions, so it has no cost at run time. The *bbox2circ* operator, defined on page 16, coerces the witness from type *bbox_signature r* to type *circ_signature r*, by dropping the information about the bounds method.

This alone is sufficient to support many object-oriented languages, in which subsumption is really just *forgetting* information about some of the methods or fields in the object. This is equivalent to so-called *width* subtyping on records. Some languages (including OCaml) support limited forms of *depth* subtyping, where the types of the fields or methods themselves can change, in a co- or contra-variant manner.

Subtyping can always be encoded using explicit coercions, but that would have a negative impact on the efficiency of our object code—unless the coercions are just



type-level operators, like the open and pack in Fig. 9. We believe it would be possible to define an inductive relation $subtype : Ty \to Ty \to \textsc{Set}$ in CIC, whose constructors implement the usual subtyping rules. A term that inhabits $subtype\ \tau_1\ \tau_2$ would thus be equivalent to a meta-logical derivation of $\tau_1 \leq \tau_2$. Our *cast* operator would be extended to accept proofs of $subtype\ \tau_1\ \tau_2$ rather than just $eq\ \tau_1\ \tau_2$. This is reminiscent of the explicit coercion techniques proposed by Crary [14], but formulating the techniques within our framework remains an avenue for future work.

### 5.2 Removing the dictionary from object representations

One of the advantages of Links, as a common IL for object-oriented languages, is its pay-as-you-go efficiency. Languages that do not need dictionaries to find method offsets at run time are not required to use them. For example, if method offsets are known at compile time, they can be hard-coded into the object types, without needing dictionaries or even symbols. Here are updates to some of the definitions from the last section.

**Definition** $FixedRep : \textsc{Set} \equiv (nat \times (Ty \to nat \to Ty))$.
**Inductive** $FixedMethod\ (r : FixedRep)\ (i : nat)\ (t : Ty) : \textsc{Set} \equiv$
$fmethod : lt\ i\ (fst\ r) \to (\Pi self : Ty.\, eq\ (snd\ r\ self\ i)\ (arw\ self\ t)) \to$
$FixedMethod\ r\ i\ t$.

We have just removed the dictionary function from the representation. The offset $i$ now appears in the *FixedMethod*, rather than remaining hidden. The signature for a circle can be expressed as follows—note the replacement of method names by method offsets:

**Definition** $circ\_fsig : FixedRep \to \textsc{Set} \equiv \lambda r.$
$(FixedMethod\ r\ 0\ fpoint \times FixedMethod\ r\ 1\ float \times FixedMethod\ r\ 2\ float)$.

The object type is the same as before, but with offsets now exposed in the bound of one of the existential quantifiers. Supporting link-time (but not run-time) use of dictionaries is more involved. If classes can be module parameters, but modules are not recursive, then all the dictionary lookups ought to be lifted to the top level in each module, outside of any loops. In this case, dictionaries should not be packaged within objects, but should just be module parameters.

### 5.3 Supporting mixins and traits

Bracha and Cook [3] define a mixin as an "abstract subclass; i.e., a subclass definition that may be applied to different super classes to create a related family of modified classes." This seems similar in spirit to the parameterized class we defined. The technical difference is that "mixins properly extend the class that they are applied to" [17]. In our example, base class methods not specified in the `CIRCLE` signature remain hidden in the derived class. In contrast, a mixin can extend an unknown base class, where any methods unspecified by the mixin are preserved in the interface of the derived class.

Following our example, a `BboxMixin` could take any class with `center` and `radius` methods, and add a `bounds` method. Any other super class methods (`area`, `move`, `enlarge`, etc.) would be preserved in the sub class. A mixin thus defines a representation *transformer* that overlays an existing dictionary with some new methods.

With simple parameterized classes, the signature can be specified as part of the definition. With mixins, this is not so simple. The signature will not be known until the point of instantiation. We do, however, need to know a few things about the transformed representation. First, it must have a bounds method, which returns a pair of points (type *frect*). Second, any methods it previously defined are *preserved.* There is one exception: if it had a bounds method previously, that one is *shadowed* by the newer definition. Thus, we must be able to say that a method label is not equal to bounds:

**Definition** $noteq : sym \to sym \to \textsc{Prop} \equiv \lambda m1\ m2.$
$\Pi k : \textsc{Set}.\Pi f\ g : k.\ ifeq\ m1\ m2\ f\ g\ =\ g.$
**Definition** $bbmix\_sig : (Rep \to \textsc{Type}) \to Rep \to \textsc{Type} \equiv \lambda sig\ r.\ \Pi r'.$
$(HasMethod\ r'\ bounds\ frect \to \Pi m\ t.\ noteq\ m\ bounds \to$
$HasMethod\ r\ m\ t \to HasMethod\ r'\ m\ t) \to sig\ r'.$

The above definition plays the role of a signature for the mixin, where the *sig* parameter is the ultimate signature, provided when the mixin is applied to a super class; $r$ is the super class representation, and $r'$ is the subclass representation.

Traits are another, similar mechanism for code reuse [29]. A trait is just a set of named methods, that can depend on some other (specified) methods. "The main difference between mixins and traits is that mixins force a linear order in their composition" [16]. We have not yet determined whether our encoding of mixins extends to traits, but we intend to pursue this as future work.

## 6 Related work

There is a long history of encoding objects and classes in typed λ-calculi and other non-object-based representations [4]. Several recent encodings are specifically designed for use in certifying compilers, where run-time efficiency is a concern [8, 13, 19, 21]. They each have their advantages—see [8] or [21] for comparisons—but none of them support separating offset determination from method retrieval.

The encoding presented in this paper is a natural generalization of the one developed by League et al. [21] for Java. They specified tuples as sequences of *rows* [28], where the tail of a sequence could be abstracted by a type variable. An object with a method in slot zero returning $\tau$ would have the type:

$$\exists \rho : Ty \to R^1.\, \mu\alpha : Ty.\, \langle \alpha \to \tau\ ;\ \rho\ \alpha \rangle$$

where the quantified variable $\rho$ conceals the types of any additional methods. Compare that to the encoding introduced in this paper:

$$\exists n : nat.\, \exists f : Ty \to nat \to Ty.\, \exists p : (0 < n \wedge (\forall \beta : Ty.\, f\ \beta\ 0 = arw\ \beta\ \tau)).$$
$$\mu\alpha : Ty.\ tup\ n\ (f\ \alpha)$$

This is the 'fixed' representation from section 5.2. In both cases, an existential hides a specification of the elements of the tuple ($\rho$ above, $f$ below), parameterized by the type of the explicit self argument. Both encodings use a recursive type in the same way: to equate the type of the self argument with the type of the object containing the methods.



Finally, both encodings reveal (in different ways) the types of known methods in the tuple.

Stone [31] developed a Calculus of Objects and Indices (COI) which has some similarities to our work. Although it is an *object* calculus (method invocation is atomic) Stone says, "it may be possible to use the ideas here to obtain a typed variant [of Links]." Like our language, COI supports dictionaries and first-class indices. Rather than singleton types, indices "have types of the form $\tau \Rightarrow \sigma$; this type classifies offsets that access a component of type $\sigma$ within an object of type $\tau$."

As specified, COI is not suitable as an intermediate language for compilers, or as a target language for proof-carrying code. It takes objects and object extension as primitive, and encodes classes in terms of objects. The class encoding does not support `super` calls, though it seems possible to add them. Due to the granularity of the calculus, optimizations like caching method pointers and devirtualization are not expressible.

Pushing COI to a lower level while maintaining soundness may be challenging. As is, its soundness relies on distinguishing between exact and inexact object types. What becomes of these concepts when objects are no longer primitive? Often, decomposing objects into tuples and functions opens up unintended ways of accessing them, leading to unsoundness [22]. It would be very interesting to see the impact of the COI design at a lower level.

## 7 Conclusion and future directions

We have developed LITL, a sound, low-level intermediate language with dictionaries, tuples, functional update, and tuple extension. Fisher et al. [17] showed that these primitives are useful for compiling various object-oriented languages, with different object models and notions of inheritance. Dictionaries support link-time or run-time determination of method offsets, for languages where the layout of a base class may not be known at compile time.

Following Shao et al. [30], the type system of LITL is embedded in the Calculus of Inductive Constructions [12]. Our reliance on CIC permits flexible reasoning about the offsets of methods, which are now first-class values with singleton types constructed from natural numbers.

We proposed a simple example in OCaml—where a super class is provided as a functor parameter—and showed by example how to encode objects, classes, method dispatch, `new`, and inheritance from a non-manifest base class. Our technique supports width (but not depth) subtyping using type coercions. Alternative representations are possible, where the dictionary is omitted (because offsets are already known) or passed separately from the object.

In the future, we expect to support depth subtyping, using a technique outlined in section 5.1. Furthermore, we intend to choose a small source language with several of these advanced object-oriented features and specify a complete type-preserving translation.

# Bibliography


[1] A. W. Appel. Foundational proof-carrying code. In *Proc. IEEE Symp. on Logic in Computer Science (LICS)*, pages 247–258, June 2001.

[2] H. Barendregt. Typed lambda calculi. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford, 1992.

[3] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 303–311, October 1990.

[4] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1–2):108–133, 1999.

[5] K. B. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good 'Match' for object-oriented languages. In *Proc. European Conf. Object-Oriented Prog.*, volume 1241 of *LNCS*, pages 104–127, Berlin, 1997. Springer-Verlag.

[6] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proc. IFIP Working Conf. on Programming Concepts and Methods*, pages 466–491, Israel, April 1990.

[7] L. Cardelli and J. C. Mitchell. Operations on records. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, Foundations of Computing Series. MIT Press, 1994.

[8] J. Chen and D. Tarditi. A simple typed intermediate language for object-oriented languages. In *Proc. Symp. on Principles of Programming Languages*. ACM, January 2005.

[9] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *Proc. Conf. on Programming Language Design and Implementation*, Vancouver, June 2000. ACM.

[10] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.0 edition, June 2004.

[11] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[12] T. Coquand and C. Paulin-Mohring. Inductively defined types. In *Proceedings of Colog '88*, volume 417 of *Lecture Notes in Computer Science*. Springer, 1990.

[13] K. Crary. Simple, efficient object encoding using intersection types. Technical Report CMU-CS-99-100, Carnegie Mellon University, Pittsburgh, January 1999.

[14] K. Crary. Typed compilation of inclusive subtyping. In *Proc. Int'l Conf. Functional Programming*, September 2000.

[15] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *Proc. Conf. on Programming Language Design and Implementation*, New York, 1999. ACM.

[16] K. Fisher and J. Reppy. A typed calculus for traits. In *Proc. Int'l Workshop on Foundations of Object-Oriented Languages*, January 2004.

[17] K. Fisher, J. Reppy, and J. G. Riecke. A calculus for compiling and linking classes. In *Proc. European Symp. on Programming*, pages 135–149, 2000.

[18] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. Conf. on Programming Language Design and Implementation*, pages 237–247, Albuquerque, June 1993.

[19] N. Glew. An efficient class and object encoding. In *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications*. ACM, October 2000.

[20] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B. Curry: Essays on Computational Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[21] C. League, Z. Shao, and V. Trifonov. Type-preserving compilation of Featherweight Java. *ACM Trans. on Programming Languages and Systems*, 24(2):112–152, March 2002.

[22] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. In G. Hedin, editor, *Proc. Int'l Conf. on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 106–120, Warsaw, April 2003. Springer.

[23] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[24] D. A. Moon. Object-oriented programming with Flavors. In *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, page 1–8, November 1986.

[25] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3), May 1999.

[26] G. C. Necula. Proof-carrying code. In *Proc. Symp. on Principles of Programming Languages*, pages 106–119, Paris, January 1997. ACM.

[27] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. Conf. on Programming Language Design and Implementation*, pages 199–208, 1988.

[28] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4, 1998.

[29] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *Proc. European Conf. Object-Oriented Programming*, July 2003.

[30] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binarios. *ACM Trans. on Programming Languages and Systems*, 27(1):1–45, January 2005.

[31] C. A. Stone. Extensible objects without labels. *ACM Trans. on Programming Languages and Systems*, 26(5):805–835, September 2004.

[32] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

**Note about the appendices:** we place here some details about our system that could not fit within the 20-page limit. Reviewers may find the details useful. If there is not room for them in the final volume, we can make them available in a technical report on the web.

# A Dynamic semantics of LITL

This section continues the dynamic semantics of LITL by distinguishing a subset of the terms as values:

$$v ::= n \mid f \mid \langle v_1, ..., v_n \rangle \mid \{l_1 = v_1, ..., l_n = v_n\} \mid [\tau_1, v \triangleright \tau_2] \mid \mathsf{fold}\, v\, \mathsf{as}\, \tau$$

The congruence reductions follow (primitive reductions are shown in section 3.2).

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 + e_2 \rightsquigarrow e_1' + e_2} \qquad \frac{e \rightsquigarrow e'}{v + e \rightsquigarrow v + e'}$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2} \qquad \frac{e \rightsquigarrow e'}{v\ e \rightsquigarrow v\ e'} \qquad \frac{e \rightsquigarrow e'}{e\,[\tau] \rightsquigarrow e'\,[\tau]} \qquad \frac{e \rightsquigarrow e'}{\mathsf{fold}\, e\, \mathsf{as}\, \tau \rightsquigarrow \mathsf{fold}\, e'\, \mathsf{as}\, \tau}$$

$$\frac{e \rightsquigarrow e'}{\mathsf{unfold}\, e \rightsquigarrow \mathsf{unfold}\, e'} \qquad \frac{e \rightsquigarrow e'}{\mathsf{cast}\,[\sigma]\, e \rightsquigarrow \mathsf{cast}\,[\sigma]\, e'} \qquad \frac{e \rightsquigarrow e'}{[\tau_1, e \triangleright \tau_2] \rightsquigarrow [\tau_1, e' \triangleright \tau_2]}$$

$$\frac{e_1 \rightsquigarrow e_1'}{\mathsf{open}\ e_1\ \mathsf{as}\ [\alpha, x]\ \mathsf{in}\ e_2 \rightsquigarrow \mathsf{open}\ e_1'\ \mathsf{as}\ [\alpha, x]\ \mathsf{in}\ e_2}$$

$$\frac{e_i \rightsquigarrow e_i'}{\langle v_1, ..., v_{i-1}, e_i, e_{i+1}, ..., e_n \rangle \rightsquigarrow \langle v_1, ..., v_{i-1}, e_i', e_{i+1}, ..., e_n \rangle} \qquad \frac{e_1 \rightsquigarrow e_1'}{e_1 \mathbin{@} e_2\,[\sigma] \rightsquigarrow e_1' \mathbin{@} e_2\,[\sigma]}$$

$$\frac{e \rightsquigarrow e'}{v \mathbin{@} e\,[\sigma] \rightsquigarrow v \mathbin{@} e'\,[\sigma]} \qquad \frac{e_1 \rightsquigarrow e_1'}{e_1 \mathbin{@} e_2\,[\sigma] \leftarrow e_3 \rightsquigarrow e_1' \mathbin{@} e_2\,[\sigma] \leftarrow e_3}$$

$$\frac{e_1 \rightsquigarrow e_1'}{v \mathbin{@} e_1\,[\sigma] \leftarrow e_2 \rightsquigarrow v \mathbin{@} e_1'\,[\sigma] \leftarrow e_2} \qquad \frac{e \rightsquigarrow e'}{v_1 \mathbin{@} v_2\,[\sigma] \leftarrow e \rightsquigarrow v_1 \mathbin{@} v_2\,[\sigma] \leftarrow e'}$$

$$\frac{e \rightsquigarrow e'}{e \fatsemi \langle e_1, ..., e_n \rangle \rightsquigarrow e' \fatsemi \langle e_1, ..., e_n \rangle} \qquad \frac{e_i \rightsquigarrow e_i'}{v \fatsemi \langle v_1, ..., v_{i-1}, e_i, e_{i+1}, ..., e_n \rangle \rightsquigarrow v \fatsemi \langle v_1, ..., v_{i-1}, e_i', e_{i+1}, ..., e_n \rangle}$$

$$\frac{e \rightsquigarrow e'}{e \# l\,[\sigma] \rightsquigarrow e' \# l\,[\sigma]} \qquad \frac{e_i \rightsquigarrow e_i'}{\{l_1 = v_1, ..., l_{i-1} = v_{i-1}, l_i = e_i, l_{i+1} = e_{i+1}, ..., l_n = e_n\} \rightsquigarrow \{l_1 = v_1, ..., l_{i-1} = v_{i-1}, l_i = e_i', l_{i+1} = e_{i+1}, ..., l_n = e_n\}}$$



# B Soundness proofs

The decidability of typing is almost immediate because the typing rules are mostly syntax directed. The places where the type derivation does not follow trivially from the syntax are:

- Rule 29 has no corresponding syntax. This does not prevent type checking from being decidable since CIC guarantees that every expression can be reduced to a normal form. We simply need to always normalize our type expressions.
- Rules 23 and 27 leave open the choice of $\tau$. This actually makes type checking undecidable. So when we type check a program we use a restriction of the above rules such that either the type of a tuple $\langle e_1, ..., e_n \rangle$ is inferred to be of the form $tup\ \widehat{n}\ (ith\ (\tau_1 :: ... :: \tau_n :: nil))$ or the programmer has to annotate the tuple with its type function $\tau$.

**Lemma 1 (Canonical forms).** *If $v$ is a value and $\circ; \circ \vdash v : \tau$, then $v$ must have the form indicated by its type:*

- $\tau =_{\beta\eta\iota} snat\ \tau_1$ *implies that* $v = n$
- $\tau =_{\beta\eta\iota} arw\ \tau_1\ \tau_2$ *implies that* $v = \lambda x{:}\tau_1.e$
- $\tau =_{\beta\eta\iota} all\ \tau_1$ *implies that* $v = \Lambda\alpha{:}\sigma.f$
- $\tau =_{\beta\eta\iota} ex\ \tau_1$ *implies that* $v = [\tau_2, v' \triangleright \tau_1]$
- $\tau =_{\beta\eta\iota} mu\ \tau_1$ *implies that* $v = \mathsf{fold}\, v'\, \mathsf{as}\, \tau_1$
- $\tau =_{\beta\eta\iota} tup\ \tau_1\ \tau_2$ *implies that* $\tau_1 = n$ *and* $v = \langle v_1, ..., v_n \rangle$
- $\tau =_{\beta\eta\iota} dict\ \tau_1$ *implies that* $v = \{l_1 = v_1, ..., l_n = v_n\}$

*Proof.* is by induction on the structure of $v$, and by adequacy of inductive definitions in an empty context for the natural number and tuple cases. □

**Theorem 1 (Progress).** *If $\circ; \circ \vdash e : \tau$ then either $e$ is a value, or there exists $e'$ such that $e \rightsquigarrow e'$.*

*Proof.* is by induction on the derivation of $\circ; \circ \vdash e : \tau$. All the cases where the toplevel subexpressions aren't simple values can be trivially reduced using the corresponding congruence rule.

**(11) variable.** Impossible case, because environment is empty.
**(12) natural number.** A numeric literal is a value.
**(13) addition.** $e = e_1 + e_2$. By induction, either $e_1$ is a value, or there exists $e_1'$ such that $e_1 \rightsquigarrow e_1'$. Likewise, either $e_2$ is a value, or there exists $e_2'$. If both are values, then they must be natural numbers (by canonical forms lemma), and we proceed with the primitive reduction for addition. Otherwise, we use the congruence rules.
**(14) functional abstraction.** $e = \lambda x{:}\sigma.e_0$. This is a value.
**(15) type abstraction.** Also a value.
**(16) application.** $e = e_1\ e_2$. Similar to addition case; by induction, either $e_1$ is a value, or there exists $e_1'$ such that $e_1 \rightsquigarrow e_1'$. If both are values, $e_1$ must have the form $\lambda x{:}\tau.e_0$ (by canonical forms lemma), so it matches the primitive reduction rule. Otherwise the inductive reduction goes through the congruence rules.

**(17) type application.** Similar.
**(18) cast.** Either goes through the congruence rule or primitive reduction of $\mathbf{cast}\,[\sigma]\,v_0$ to $v_0$. (Trivial.)
**(19) existential introduction.** $e = [\tau', e_0 \triangleright \tau']$. Either $e_0$ is a value, in which case so is the package, or $e_0$ can be reduced, in which case we apply the reduction through the package congruence rule.
**(20) existential elimination.** Similar to application and type application, including use of canonical form of existential value.
**(21) fold.** Becomes a value if the sub-expression is a value, or goes through fold congruence rule.
**(22) unfold.** Go through unfold congruence rule, or if sub-expression is a value, it must be a fold (due to canonical forms lemma) in which case the primitive reduction matches.
**(23) tuple.** Either is a value, or goes through one of the congruence rules.
**(24) tuple selection.** Two congruence rules are available. If both sub-expressions are values then we need several prerequisites to use the primitive reduction. First, the left-hand side must be a tuple value of length $n$ (by canonical forms). Next, the right-hand side must be a natural number (by canonical forms). Finally, the index must be less than the length. Here we rely on the adequacy of arithmetic and *lt* in an empty context. Follow the arguments in TSCB paper.
**(25) functional update.** Similar to previous case.
**(26) tuple extension.** Canonical forms guarantees the left side is a tuple, so the primitive reduction applies.
**(27) dictionary construction.** Either a value or use a congruence rule.
**(28) dictionary lookup.** If $e$ is not a value, we use the congruence rule. Otherwise, by canonical forms $e$ has to be a dictionary. By the typing rule of the dictionary constructor, we know that the dictionary typing function $\tau$ returns *some* $\tau_i$ iff applied to one of the labels in the dictionary. Since $\sigma$ is a proof that $\tau$ returns *some* $\tau'$, it follows that $l$ is indeed one of the $l_i$ of the dictionary and the primitive reduction applies.
**(29) type conversion.** Trivial: the inductive hypothesis already gives us our conclusion. □

**Lemma 2 (Substitution).**
*If* $\Delta; \Gamma, x{:}v \vdash e : \tau$ *then* $\Delta; \Gamma \vdash e[v/x] : \tau$.
*If* $\Delta, \alpha{:}\tau; \Gamma \vdash e : \tau'$ *then* $\Delta; \Gamma[\tau/\alpha] \vdash e[\tau/\alpha] : \tau'[\tau/\alpha]$.

*Proof.* is straightforward, by induction on the typing derivation. □

**Theorem 2 (Subject reduction).** *If* $\circ; \circ \vdash e : \tau$ *and* $e \rightsquigarrow e'$, *then* $\circ; \circ \vdash e' : \tau$.

*Proof.* is by induction on the derivation of $e \rightsquigarrow e'$. All the congruence rules are proved trivially from the induction hypothesis because they all reduce the subexpression in the same empty context.

**(1) addition.** The typing rule of the redex is #13, so $\tau =_{\beta\eta\iota} \mathit{snat}\ (\mathit{plus}\ \tau_1\ \tau_2)$. So we need to show that $n_3$ has that type, using rule #12.



**(2) beta reduction.** The typing derivation of the redex uses rule #16 preceded by #14, and $\tau =_{\beta\eta\iota} \mathit{arw}\ \tau_1\ \tau_2$. We use the value substitution lemma.
**(3) type application.** Same situation except we use the type substitution lemma.
**(4) cast.** This is a critical case. We know $\mathbf{cast}\,[\_]\,e$ has type $\tau_2$, and $e$ has type $\tau_1$. This follows from the fact that we know '$\mathit{eq}\ \tau_1\ \tau_2$' and that in an empty context this can only be true if $\tau_1 =_{\beta\eta\iota} \tau_2$ so we can use the typing rule #29.
**(5) open.** This uses both substitution lemmas.
**(6) unfold.** Trivial.
**(7) update.** Trivial as well.
**(8) extend.** We can prove that $\mathit{eq}\ \mathit{append}\ \tau_1\ \tau_2\ \tau_2'\ \widehat{i}\ \tau_2\ \widehat{i}$ for all $i$ smaller than $\tau_1$, and that it is equal to $\tau_2'\ \widehat{i}$ otherwise. The rest follows trivially, except that we need to use the typing rule #29 to account for the fact that we only know equality in terms of *eq*, as was the case for cast.
**(9) select.** Trivial.
**(10) lookup.** Straightforward since the core of the proof is provided as an annotation. □

## C Proofs needed to extend an unknown base class

This is an extended development, with Coq proofs, of the reasoning in section 4.6. Our goal there was to prove that a *circ_signature* representation, extended with *bbox_rep*, matches the *bbox_signature*. This depends critically on the semantics of *append*.

Specifically, extending a tuple with new elements does not alter the types of the existing elements. We will use Coq tactics to prove this, but the resulting proof can be expressed as a normal term in CIC. The proof refers to *lt_S_n*, a lemma in the Coq library stating that if $S\,n < S\,m$ then $n < m$.

```
Lemma append_semantics1 : Π i n. lt i n → Π f g. eq (append n f g i) (f i).
Proof.
 induction i. induction n.
   intro H; inversion_clear H. intros _ f g; apply (refl_equal (f 0)).
  induction n.
   intro H; inversion_clear H. intro H; assert (lt i n).
    apply lt_S_n; assumption. intros f g; exact (IHi n H0 (λx. f (S x)) g). □
```

The following simple lemma will express the same result in a more useful form, so that it matches one of the properties required by *HasMethod*.

```
Lemma extension_okay : Π i n. lt i n → Π f t.
  (Π s. eq (f s i) (arw s t)) → Π g self. eq (append n (f self) (g self) i) (arw self t).
Proof.
 intros i n lt f t p g self. assert (H1 ≡ p self).
 assert (H2 ≡ append_semantics1 lt (f self) (g self)). exact (trans_eq H2 H1). □
```

With this result, we can take information about a base class tuple, and transform it into information about a derived class tuple, to which other methods have been appended.

We will also need to extend the *lt* proofs within *HasMethod*. For a given offset (*i*), known to be less than the size of the parent tuple (*n*), it is also of course less than the size of the extended tuple:

```
Lemma lt_plus_bound : Π i n k. lt i n → lt i (plus k n).
Proof.
 intros i n k H. assert (L ≡ lt_plus_trans i n k H).
 rewrite (plus_comm k n). assumption. □
```

This was a simple corollary of *lt_plus_trans* in the Coq library, whose result is commutative (*plus n k*).

These lemmas have helped us prove things about inherited methods. To prove anything about new methods (such as bounds), we will need another lemma about the semantics of *append*. It describes what happens when the index is $\geq n$.

```
Lemma append_semantics2 : Π k n f g. eq (append n f g (plus k n)) (g k).
Proof.
 induction k. induction n.
   intros f g; exact (refl_equal (g 0)).
   intros f g; exact (IHn (λx. f (S x)) g).
  induction n.
  intros f g; exact (f_equal g (plus_0_r (S k))).
  intros f g. assert (eq
       (append n (λx. f (S x)) g (plus k (S n)))
       (append n (λx. f (S x)) g (plus (S k) n))).
   apply (f_equal (append n (λx. f (S x)) g) (sym_eq (plus_Snm_nSm k n))).
   apply (trans_eq H (IHn (λx. f (S x)) g)). □
```

Again, with transitivity of equality, we coerce this into a more usable form.

```
Lemma extension_effect : Π k g t. (Π self. eq (g self k) (arw self t)) →
  Π n f self. eq (append n (f self) (g self) (plus k n)) (arw self t).
Proof.
 intros k g t p n f self. assert (L ≡ append_semantics2 k n (f self) (g self)).
 assert (M ≡ p self). exact (trans_eq L M). □
```

Finally, we can prove that a representation matching *circ_signature* can be extended by *bbox_rep* to a representation matching *bbox_signature*. To show how this proof may be adapted to other class signatures, we have defined tacticals for the two kinds of cases: inherited methods and new methods.

```
Definition bbox_witness : Π r. Π p : circ_signature r. bbox_signature (bbox_rep p).
Proof.
 let inherit ≡ λname ty sel.
   apply (method (bbox_rep p) name
     (lt_plus_bound 1 (proof (sel r p))) (refl_equal (Some (snat (offset (sel r p)))))
     (extension_okay (proof (sel r p)) (tupfn r)
       (tupeq (sel r p)) (λs. ith _))) in
```



```
 let add ≡ λname ty k pf.
   apply (method (bbox_rep p) name
     (plus_lt_compat_r k 1 (size r) pf) (refl_equal (Some (snat (plus k (size r)))))
     (extension_effect k
       (λs. ith (arw s frect :: nil))
       (λs. refl_equal (arw s ty))
       (size r) (tupfn r))) in
(repeat constructor;
 [ inherit center fpoint circ_center | inherit radius float circ_radius
 | inherit area float circ_area | add bounds frect 0 (le_n 1)]). □
```

The *inherit* and *add* tacticals are specific to the *bbox* extension only where they include the literal 1 (representing the number of methods added by *bbox*) and refer to the types of the new methods (*arw s frect*). This is important because, in practice, a compiler would produce this proof. It must be automatically derivable from the base and derived class signatures.

## D Representing symbols

We used symbols in CIC throughout this work without properly defining them. Method labels in the source programming language could be mapped to natural numbers, but here we show how to define them as sequences of characters from some alphabet.

```
Inductive char : SET ≡
 A | B | C | D | E | F | G.
Definition sym : SET ≡ list char.
```

To encode the LITL semantics, the only operation we need on symbols is equivalence:

```
Definition ifeqc ≡ λx y : char. λk : SET. λt f : k.
 match x, y with
 | A, A ⇒ t | B, B ⇒ t | C, C ⇒ t | D, D ⇒ t
 | E, E ⇒ t | F, F ⇒ t | G, G ⇒ t | _, _ ⇒ f
 end.
Fixpoint ifeq (x y : sym) (k : SET) (t f : k) {struct x} : k ≡
 match x, y with nil, nil ⇒ t
 | c :: cs, d :: ds ⇒ ifeqc c d (ifeq cs ds t f) f
 | _, _ ⇒ f
 end.
```

Here are some (abbreviated) method names used in examples.

```
Definition center ≡ C :: E :: nil.
Definition radius ≡ A :: D :: nil.
Definition area ≡ A :: E :: nil.
Definition bounds ≡ B :: D :: nil.
```

## E Encoding terms in Coq

Throughout the paper, the type language of LITL is expressed in Coq notation and automatically checked for validity. The term language of LITL, so far, does not benefit from this approach. To demonstrate that the LITL terms shown here are indeed type-correct, we can encode their static semantics within Coq. Here as an inductive definition of type-indexed terms:

```
Inductive Exp : Ty → SET :=
 | enat : Π n. Exp (snat n)
 | eadd : Π n m. Exp (snat n) → Exp (snat m) → Exp (snat (plus n m))
 | eabs' : Π (R : Ty → SET) (t v : Ty). (R t → Exp v) → Exp (arw t v)
 | etabs : Π (k : SET) (s : k → Ty). (Π j : k. Exp (s j)) → Exp (all s)
 | eapp : Π s t : Ty. Exp (arw s t) → Exp s → Exp t
 | etapp : Π (k : SET) (s : k → Ty). Exp (all s) → Π t : k. Exp (s t)
 | ecast : Π s t : Ty. eq s t → Exp s → Exp t
 | epack : Π (s0 : SET) (t1 : s0 → Ty) (t0 : s0). Exp (t1 t0) → Exp (ex t1)
 | eopen' : Π R : Ty → SET. Π s0 : SET. Π t1 : s0 → Ty. Π t2 : Ty. Exp (ex t1) →
     (Π a : s0. R (t1 a) → Exp t2) → Exp t2
 | efold : Π (s : Ty → Ty). Exp (s (mu s)) → Exp (mu s)
 | eunfd : Π (s : Ty → Ty). Exp (mu s) → Exp (s (mu s))
 | etup : Π (n : nat) (ts : list Ty). Es n ts → Exp (tup n (ith ts))
    (* The constructor above is more restrictive than the typing rules, *)
    (* but it ensures we stick to a decidable subset. *)
 | esel : Π (j n : nat) (f : nat → Ty). Exp (tup n f) → Exp (snat j) → lt j n → Exp (f j)
 | eupd : Π (j n : nat) (f : nat → Ty). Exp (tup n f) → Exp (snat j) → Exp (f j) →
     lt j n → Exp (tup n f)
 | eext : Π (n n' : nat) (f f' : nat → Ty). Exp (tup n f) → Exp (tup n' f') →
     Exp (tup (plus n' n) (append n f f'))
 | edict : Π m : map. Ds m → Exp (dict (lookup m))
 | elook : Π (g : sym → option Ty). Exp (dict g) →
     Π (s : sym) (t : Ty). eq (g s) (Some t) → Exp t
 | efix' : Π (R : Ty → SET) (t v : Ty). (R (arw t t) → R t → Exp v) → Exp (arw t v)
 | ecmp : Π n m. Exp (snat n) → Exp (snat m) →
     Exp (snat (if (beq_nat n m) then 1 else 0))

with Es : nat → list Ty → SET := enil : Es O nil
 | econs : Π (t : Ty) (n : nat) (ts : list Ty). Exp t → Es n ts → Es (S n) (t :: ts)

with Ds : map → SET := dnil : Ds nil
 | dcons : Π (s : sym) (t : Ty) (m : map). Exp t → Ds m → Ds ((s,t) :: m).
```

The following are for notational convenience:

```
Definition eabs := eabs' Exp.
Definition efix := efix' Exp.
Implicit Arguments eabs [v].
```



```
Definition eopen := eopen' Exp.
Definition elet := λ s t : Ty. λ e : Exp s. λ body : Exp s → Exp t. eapp (eabs s body) e.
Definition dcons' := λ t m (x : sym × Exp t) (xs : Ds m).
  dcons (t ≡ t) (m ≡ m) (fst x) (snd x) xs.

Notation "'λ' x : t. e" := (eabs t (λ x. e)) (at level 200, x ident).
Notation "'Open' x y = e1 'in' e2" := (eopen e1 (λ x y. e2))
  (at level 200, x ident, y ident).
Notation "'Let' x = e1 'in' e2" := (elet e1 (λ x. e2)) (at level 200, x ident).
Notation "'Λ' x : t. e" := (etabs (λ x : t. _) (λ x. e)) (at level 200, x ident).
Notation "'⟨' x , .. , y '⟩'" := (etup (econs x .. (econs y enil) ..)).
Notation "x '↦' y" := (x, y) (at level 100).
Notation "{ x , .. , y }" := (edict (dcons' x .. (dcons' y dnil) ..)).
Notation "≪ w , e | t ≫" := (epack t w e) (at level 200).
Notation "e1 @ e2 [ t ]" := (esel e1 e2 t) (at level 99).
Notation "e # l [ t ]" := (elook e l t) (at level 99).
```

Now we are ready to express the running examples within Coq. Here is the encoding of the function that invokes the radius method on a circle, from figure 5:

```
Definition invoke_radius : Exp (arw (objty circ_signature) float) ≡
  λ x : objty circ_signature.
    Open r x1 = x in
    Open p x2 = x1 in
    Let x3 = eunfd x2 in
    Let dc = x3 @ enat 0 [lt02] in
    Let vt = esel x3 (enat 1) lt12 in
    match p with (_, pr, _) ⇒
     Let j = elook dc radius (dicteq pr) in
     Let fp' = esel vt j (proof pr) in
     Let fp = ecast (tupeq pr (selfty r)) fp' in
     eapp fp x2
    end.
```

And the function to create a new circle, from figure 6:

```
Definition lt03 : lt 0 3 ≡ le_S (le_S (le_n 1)).
Definition lt13 : lt 1 3 ≡ le_S (le_n 2).
Definition lt23 : lt 2 3 ≡ le_n 3.
Definition new_circ : Exp (arw (classty circ_signature) (objty circ_signature)) ≡
  λ c0 : classty circ_signature.
    Open r c1 = c0 in
    Open p c2 = c1 in
    Let dc = esel c2 (enat 1) lt13 in
    Let ms = esel c2 (enat 2) lt23 in
    Let vt = etapp (etapp ms r) p in
```

```
  Let x = ⟨dc, vt⟩ in
  epack (objty′ circ_signature) r
   (epack (objty″ circ_signature r) p
    (efold (objrep r) x)).
```

Here is the 'B' circle class, to demonstrate that the *classty* is habitable (figure 7).

```
Parameter method_body : Π t : Ty. Exp t.
Definition circB : Exp (classty circ_signature) ≡
  Let dc = {radius ↦ enat 4, area ↦ enat 1,
        center ↦ enat 2} in
  Let ms = Λ r : Rep. Λ p : circ_signature r.
        ⟨λ s : selfty r. method_body (ex snat),
         λ s : selfty r. method_body float,
         λ s : selfty r. method_body fpoint,
         enat 0,
         λ s : selfty r. method_body float⟩ in
  epack (classty′ circ_signature) circB_rep
   (epack (classty″ circ_signature circB_rep) circB_witness
    ⟨enat 5, dc, ms⟩).
Definition create_and_invoke : Exp float ≡
  eapp invoke_radius (eapp new_circ circB).
```

The following corresponds to figure 8.

```
Parameter area_formula : Exp (arw float float).
Definition circle_bbox :
Exp (arw (classty circ_signature) (classty bbox_signature)) ≡
  λ c0 : classty circ_signature.
    Open r0 c0 = c0 in
    Open p0 c0 = c0 in
    Let sz0 = esel c0 (enat 0) lt03 in
    Let dc0 = esel c0 (enat 1) lt13 in
    Let ms0 = esel c0 (enat 2) lt23 in
    Let ci = elook dc0 center (dicteq (circ_center p0)) in
    Let ri = elook dc0 radius (dicteq (circ_radius p0)) in
    Let ai = elook dc0 area (dicteq (circ_area p0)) in
    Let dc1 = {center ↦ ci, radius ↦ ri, area ↦ ai, bounds ↦ sz0} in
    let r1 ≡ bbox_rep p0 in
    Let ms1 = Λ r2 : Rep. Λ p2 : bbox_signature r2.
      Let vt0 = etapp (etapp ms0 r2) (bbox2circ p2) in
      Let ar = esel vt0 ai (proof (circ_area p0)) in
      Let ar = ecast (tupeq (circ_area p0) (selfty r2)) ar in
      Let ar′ = λ s : selfty r2. eapp area_formula (eapp ar s) in
      Let ar′ = ecast (sym_eq (tupeq (circ_area p0) (selfty r2))) ar′ in
      Let bo = λ s : selfty r2. method_body frect in
      Let vt1 = eupd vt0 ai ar′ (proof (circ_area p0)) in
```



```
      eext vt1 ⟨bo⟩ in
    epack (classty′ bbox_signature) r1
     (epack (classty″ bbox_signature r1) (bbox_witness p0)
      ⟨eadd (enat 1) sz0, dc1, ms1⟩).
```

The upcast in figure 9:

```
Definition bbox_upcast :
Exp (arw (objty bbox_signature) (objty circ_signature)) ≡
  λ x : objty bbox_signature.
    Open r x = x in
    Open p x = x in
    epack (objty′ circ_signature) r
     (epack (objty″ circ_signature r) (bbox2circ p) x).
```

All of these examples were extracted from the paper and successfully validated by Coq.