Cayenne — a language with dependent types

Lennart Augustsson

Department of Computing Sciences
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Email: augustss@cs.chalmers.se
WWW: http://www.cs.chalmers.se/~augustss

Abstract. Cayenne is a Haskell-like language. The main difference between Haskell and Cayenne is that Cayenne has dependent types, i.e., the *result type* of a function may depend on the *argument value*, and types of record components (which can be types or values) may depend on other components. Cayenne also combines the syntactic categories for value expressions and type expressions; thus reducing the number of language concepts.
Having dependent types and combined type and value expressions makes the language very powerful. It is powerful enough that a special module concept is unnecessary; ordinary records suffice. It is also powerful enough to encode predicate logic at the type level, allowing types to be used as specifications of programs. However, this power comes at a cost: type checking of Cayenne is undecidable. While this may appear to be a steep price to pay, it seems to work well in practice.
Keywords: Type systems, language design, dependent types, module systems

1 Introduction

Languages like Haskell [Hud92] and SML [MTH90] have type systems that are among the most advanced of any language. Despite this there are things that are inexpressible in these type systems. Dependent types, i.e., having types depend on values, increases the expressiveness of type systems and many of the problems of Hindley-Milner typing can be overcome.

Cayenne is a Haskell-like language that combines dependent types and first class types, i.e., types can be used like values. The syntax for value and type expressions is the same. Cayenne does not have a separate notion of modules; records are used as modules, this means that the language for combining modules is also the usual expression language. This is in contrast with Haskell and SML. Haskell has similar but different syntax for type and value expressions and definitions. SML has different syntax for value, type, and module expressions and definitions. It can be argued that they should look different, because they are different. But we want to argue the opposite, the facilities for the three types of expressions are similar, so why should they be different? In Cayenne they are

the same and exactly the same program constructs can be used on all levels, thus reducing the number of concepts that you need to master.

Although dependent types have been used before in proof systems, e.g., [CH88], to our knowledge this is the first time that the full power of dependent types has been integrated into a *programming language*.

We will now give some motivating examples, where we show problems in Haskell that are solved in Cayenne. The differences between Haskell and Cayenne will be explained as they occur.

1.1 The type of `printf`

The C standard I/O library has a very useful function for doing output, namely `printf`. The function `printf` takes a formatting string as the first argument and then some additional arguments. The number of arguments and their types depends on the formatting string. It is simple to write a similar function in Haskell, but it will not type check.[1]

```
printf fmt = pr fmt "" where
  pr ""           res = res
  pr ('%':'d':cs) res =
      \ i -> pr cs (res ++ show (i::Int))
  pr ('%':'s':cs) res =
      \ s -> pr cs (res ++ s)
  pr ('%': c :cs) res =
      pr cs (res ++ [c])
  pr (c:cs)       res =
      pr cs (res ++ [c])
```

This is a very simplified version of `printf`, but as in the real version, the substring "`%d`" marks an integer argument and "`%s`" marks a string argument. The type of `printf` clearly varies with its first argument; e.g.,

```
printf "%d" :: Int -> String
printf "%s owes %d SEK to %s" ::
    String -> Int -> String -> String
```

As we can see, the function is easy to write and works perfectly, but cannot be given a type in Haskell.[2]

Cayenne solution The type of `printf` can easily be computed from the first argument. All we need to do is to write a function that computes the right type. The type of all types is called "`#`"[3] in Cayenne.

[1] The code given here is very inefficient, but that is easy to remedy.

[2] Olivier Danvy has recently shown, [Dan98], that functions similar to `printf` can be given a type with Hindley-Milner typing with a clever trick.

[3] We would like to use the more familiar notation "`*`" for the type of types. This might be possible, but it interacts badly with the use of "`*`" as an infix operator.

```
PrintfType :: String -> #
PrintfType ""           = String
PrintfType ('%':'d':cs) = Int    -> PrintfType cs
PrintfType ('%':'s':cs) = String -> PrintfType cs
PrintfType ('%': _ :cs) =           PrintfType cs
PrintfType ( _ :cs)     =           PrintfType cs

printf :: (fmt::String) -> PrintfType fmt
printf fmt = pr fmt ""

pr :: (fmt::String) -> String -> PrintfType fmt
pr ""           res = res
pr ('%':'d':cs) res =
      \ (i::Int)    -> pr cs (res ++ show i)
pr ('%':'s':cs) res =
      \ (s::String) -> pr cs (res ++ s)
pr ('%': c :cs) res =
      pr cs (res ++ [c])
pr (c:cs)       res =
      pr cs (res ++ [c])
```

The function `PrintfType` mimics the recursive structure of `printf`, but it computes the type instead of the value. E.g.,
`PrintfType "%d"` $\longmapsto$ `Int -> String`
The typing of `printf` is now
`printf :: (fmt::String) -> PrintfType fmt`
This example differs from Haskell in that the first argument (which has type `String`) has a name, `fmt`, which can be used in the type expression. A minor point to note is that λ-expressions in Cayenne have an explicit type on the bound variable, whereas they do not in Haskell.

Another example of a function with a dependent type can be found in appendix B.

1.2 The set "package"

Record types in Haskell (and SML) can contain values, but not types; sometimes this can be inconvenient. To show an example of this we will use a simple set of integers. It should support creating the empty set, the singleton set, taking union, and testing for set membership. There are many possible ways to implement these sets and sometimes you want to have multiple implementations in a program and choose dynamically which one to use (e.g., depending on the use pattern). To be able to do this we would like to be able to store different set implementations in a data structure.

We would want to define the set type something like

```
data IntSet = IntSet {
    type T,
    empty :: T,
    singleton :: Int->T,
    union :: T->T->T,
    member :: Int->T->Bool
    }
```

Unfortunately, this is not possible since we cannot have a type in a record and the name `T` would also not be in scope. This kind of construct is only available at the module level in Haskell, but modules are definitely not first class objects in Haskell; there are no operations on modules except for the importation of them. SML allows this kind of definitions on the module level and has a rich language for combining them, but they are still not first class objects, so they cannot be put in a run-time data structure.[4]

Cayenne solution Cayenne records are different from Haskell records in several respects: Cayenne records are not data types, they can contain types, and when defining a record object the labels are bound within the record expression. The `sig` keyword starts a record type and the `struct` keyword starts a record value.

The `IntSet` type could be defined like this

```
type IntSet = sig
    type T
    empty :: T
    singleton :: Int->T
    union :: T->T->T
    member :: Int->T->Bool
```

An implementation could look like this

```
naïveSet :: IntSet
naïveSet = struct
    abstract type T = Int->Bool
    empty x = False
    singleton x x' = x == x'
    union s t x = s x || t x
    member x s = s x
```

This kind of record borrows features from Haskell modules, but they are still first class objects.

1.3 The `Eq` class

The `Eq` class in Haskell has the following definition:[5]

[4] It is not obvious that the first class modules proposed here extend easily to a language like SML that supports side effects.

[5] It also has a definition of `(/=)`, but it is of no use in this example so we disregard it.

```
class Eq a where
    (==) :: a -> a -> Bool
```

This, quite correctly, states that `(==)` takes two arguments of the same type and returns a boolean, but surely this is not all we expect from an equality. We expect it to be a "real" equality, i.e., we most likely want it to be an equivalence relation.[6] The equivalence property of equality cannot be expressed in Haskell. The best we can do is to have it as a comment, and hope that each equality defined in the program is really an equivalence relation.

Cayenne solution Cayenne has no type classes so the `Eq` class problem must be reformulated slightly. A class definition in Haskell would correspond to a type definition of a record in Cayenne, and instance declarations in Haskell correspond to values of that type. All dictionaries will thus be passed explicity in Cayenne.

The `Eq` "class" in Cayenne would be

```
type Eq a = sig
    (==) :: a -> a -> Bool
```

To include an equivalence proof we must first have a way of expressing logical properties. This is, in fact, easy since Cayenne types can, through the Curry-Howard isomorphism, encode predicate calculus as types, see figure 1. Terms of the different types correspond to the proof of the corresponding properties. This is all well known from constructive type theory [NPS90], and well before that [How80].

Predicate calculus	Cayenne type
$\bot$	`Absurd` (or any empty type)
$\top$	any non-empty type
$x \vee y$	`Either` x y
$x \wedge y$	`Pair` x y
$\forall x \in A.P(x)$	`(`x`::`A`) -> `$P(x)$
$\exists x \in A.P(x)$	`{`x`::`A`; `y`::`$P(x)$`}`

```
data Absurd =
data Pair x y = pair x y
data Either x y = Left x | Right y
```

Fig. 1. "Encoding" predicate logic as Cayenne types.

We encode the absurd proposition (i.e., falsity) by the empty type, and all types with elements encode truth. The dependent function type encodes universal quantification and records encode existential quantification. Proving a property correspond to finding an element (i.e., constructing a value) in a type.

[6] Or, even better, a congruence relation.

Since false logical statements correspond to the empty type we cannot find any values in them, but in (constructively) true logical statements we can.

One way of solving our problem in Cayenne is to extend the `Eq` type like this:

```
type Eq a = sig
    (==) :: a -> a -> Bool
    equiv :: Equiv (LiftBin (==))
```

`LiftBin` is a function that maps a binary operation yielding a Boolean into a corresponding relation. `Equiv` is a predicate on relations stating that the relation is an equivalence relation.

The following auxiliary definitions are used above. Further differences between Haskell and Cayenne appear below: type variables must be bound, but are often used as *hidden* arguments, introduced by the `|->` function arrow, see section 3.1 for further discussion.

```
data Absurd =

data Truth = truth

Lift :: Bool -> #
Lift (False) = Absurd
Lift (True)  = Truth

LiftBin :: (a :: #) |-> (a -> a -> Bool) -> Rel a
LiftBin |a op = \ (x::a) -> \ (y::a) -> Lift (op x y)

type Rel a = a -> a -> #

Refl :: (a :: #) |-> Rel a -> #
Refl |a R = (x::a) -> x `R` x

Symm :: (a :: #) |-> Rel a -> #
Symm |a R = (x::a) -> (y::a) -> x `R` y -> y `R` x

Trans :: (a :: #) |-> Rel a -> #
Trans |a R = (x::a) -> (y::a) -> (z::a) ->
    x `R` y -> y `R` z -> x `R` z

Equiv :: (a :: #) |-> Rel a -> #
Equiv R = sig
    refl :: Refl R
    symm :: Symm R
    trans :: Trans R
```

Appendix A contains the complete code for this example with some instances.

2 Core Cayenne

Cayenne has three basic type forming constructs: dependent functions, data types (sums), and dependent records (products).[7] Core Cayenne is the subset of Cayenne that has no syntactic bells and whistles, just the basic constructs. We will start by looking at Core Cayenne and then at the various syntactic shorthands. The syntax of Core Cayenne is given in figure 2. The grammar disregards certain minor concrete syntax issues. There is no syntactic distinction between expressions and types in Cayenne, as is reflected in the grammar.

expr	::=	`(` *varid* `::` *type* `)` `->` *expr*	function type
		`\` `(` *varid* `::` *type* `)` `->` *expr*	λ expression
		expr expr	application
		`data` { *conid* { *type* } | }	sum type
		conid `@` *type*	constructor
		`case` *varid* `of` { *arm* } `::` *type*	sum scrutinization
		`sig` { *sign* }	record type
		`struct` { *defn* }	record formation
		expr `.` *lblid*	record selection
		id	variable
		$\#_n$	type of types
arm	::=	`(` *conid* { *varid* } `)` `->` *expr* `;`	
		varid `->` *expr* `;`	
sign	::=	*lblid* `::` *type* `;`	
		lblid `::` *type* `=` *expr* `;`	
defn	::=	*vis lblid* `::` *type* `=` *expr* `;`	
vis	::=	`private` | `public` *abs*	
abs	::=	`abstract` | `concrete`	
type	::=	*expr*	
varid	::=	*id*	
conid	::=	*id*	
lblid	::=	*id*	

Fig. 2. Core Cayenne abstract syntax grammar. Metasyntax: { } are used to denote repetition of an arbitrary number of items.

2.1 Functions

Function expressions are written as λ-expressions. The bound variable must be given a type. The function type is written like the λ-expression, but without the leading "\".

[7] The terminology is a little confusing here, what in constructive type theory is usually called dependent products is called dependent functions in this paper and what in CTT is called dependent sums is called dependent records here. The latter terminology is more in the tradition of programming languages.

The big difference between the Cayenne function type and the Haskell function is that since the bound variable is available to the right of the arrow, the result type of a function may depend on the value of the argument.

Function application is written with juxtaposition as usual.

Example:

```
\ (x::Int) -> inc x
```

which has type

```
(x::Int) -> Int
```

2.2 Data types

Unlike Haskell, a data type (sum type) does not have to be given a name; there is an expression that denotes each data type. E.g., "`data False | True`" is the type of booleans.

Constructors are written in a way that is very different from Haskell. The constructor names used in a data type expression have no name restrictions (unlike Haskell where they have to be capitalized) and need not be unique. Consequently, given only the name of a constructor it is impossible tell what type it constructs. Therefore, constructors are given with their types in Cayenne. E.g., "`True@(data False | True)`" is one of the constructors for the boolean type, or, if "`Bool`" has been defined, it can be written "`True@Bool`". Constructor names are not part of the usual name space; they can only occur in "`@`"-expressions and `case` expressions and in the latter the type that they construct can be deduced.

Case expressions in Core Cayenne look a little different from Haskell. Only simple patterns are allowed and all constructor patterns have to be parenthesised to distinguish them from variable patterns. Apart from the scrutinized variable and the case arms, the case expression also has a type attached. This type expression gives the type of the arms of the case expression. Note that this expression can contain the scrutinized variable so the type may depend on it. The reason for having this type is that with dependent types it is not in general possible to figure out the type of the case expression.

Example:

```
case l of
(Nil) -> True;
(Cons x xs) -> False;
   :: Bool
```

An example with a dependent type:

```
case l of
(True)  -> 1;
(False) -> "Hello";
   :: (case l of (True)  -> Int;
                 (False) -> String)
```

2.3 Records

The record type (product type) in Cayenne is the most complicated of the type formers. The reason for this is that records also serve the purpose of modules in most other languages.

A record type is written as `sig` followed by a signature for each component of the record. The signature normally gives only the type of the component, but it can also give the *value* of it. This feature is sometimes called a translucent sum, and is described in more detail in section 4.1.

A record is formed by the `struct` keyword followed by bindings for all the record components. Each binding gives the type and value of the component as well as its visibility. The names of the record components (the labels) are in scope within the record expression. This means that the bindings are mutually recursive.[8]

The visibility for a record component determines how it will show up in the type of the record. A `private` component does not show up at all in the type of the record, a `public abstract` component has only its type, and a `public concrete` component has both its type and value in the type of the record.

A record component, which occurs in (i.e., which is not `private`) the type of the record, can be extracted with the usual dot notation.

Examples:

```
struct
   private x :: Int = 1
   public abstract y :: Int = x+1
   public concrete z :: Int = x+2
```

has type

```
sig
   y :: Int
   z :: Int=3
```

Selection: `r.y + r.z`

2.4 The type of types

The type of types is $\#_1$, this type has type $\#_2$ which has type $\#_3$ etc. The reasons for using a stratified type system are twofold: first, using "`# :: #`" would, even in the absence of recursion, make the Cayenne type system unsound as a logic as it would allow Girard's paradox; second, the unstratified type system would make it impossible during type checking to determine if an expression corresponds to a type or a real value and it would be impossible to remove the types at runtime, see section 6.1.

Note that there is no elimination construct for the `#` type, i.e., no `casetype` construct. It would be possible and useful to have such a construct, but Cayenne currently lacks it, partly because having it would make it impossible to remove runtime type information, see 6.1.

[8] Though there are restrictions on how the recursion may occur in the signatures to ensure that the type can be viewed as a fixpoint of a Σ-type.

3 Full Cayenne

Using Core Cayenne would be feasible, but quite tedious, just like using the bare λ-calculus is. Cayenne has many syntactic constructs to make it more palatable and closer to an ordinary functional language.

3.1 Hidden arguments

Many functions have type arguments that seem to serve no purpose, except to irritate the user. E.g.

```
if :: (a :: #) -> Bool -> a -> a -> a
```

for each use of `if` the type of the two branches must be given as the first argument.

To lessen this problem Cayenne uses a mechanism for leaving out certain arguments at the application site. However, the arguments still must be given when the function is defined. Hidden arguments introduce a new version of the function type, the function abstraction, and the function application.

The function arrow in both the type and abstraction notation is written `|->` for hidden arguments. Application of a hidden argument uses infix `|`, but normally a hidden argument does not need to be given at all.

In function definitions the hidden arguments should not be present on the left hand side unless preceded by a `|`, i.e., the left hand side looks like an application.

Example:

```
if :: (a :: #) |-> Bool -> a -> a -> a
if (True)  x y = x
if (False) x y = y
```

This "`if`" function can the be used as "`if True 1 2`", or more explicitly "`if |Int True 1 2`".

The concept of hidden arguments is a syntactic device without any deep semantic properties. The function type for hidden arguments should not be viewed as a new type. It is completely compatible with the normal function type. It only serves as a marker to aid the insertion of the hidden arguments. This view of hidden arguments was presented in [ACN90] and later used in other systems like Lego, [LP92], where the concept was formalized. Similar mechanism exist e.g., in Quest, [Car94], and Russell, [BDD89].

The current implementation of hidden arguments is quite weak and cannot always find the hidden arguments even when it seems reasonable that it should. It can find a hidden argument if the variable (`a` in the example) occurs in a later argument type or the result type. In the future we will probably switch to a more powerful method that introduces metavariables (in the sense of logical frameworks) and tries to derive their values using more powerful methods such as unification.

3.2 Syntactic sugar

This is a brief list of syntactic extensions that can be regarded as mere "sugar".

- If the variable bound in the function type does not occur anywhere it can be dropped and the function type is thus written as in Haskell. E.g. "`(x::Int)->Int`" can be written as "`Int->Int`" instead.
- Infix operators (with a fixed set of precedences) can be used. The same conventions as in Haskell are used.
- The patterns in case arms can be written in the normal Haskell style with nested patterns etc. The type part of case expression is only necessary if the type of the right hand sides depend on the scrutinized expression.
- `public` can be omitted, since it is the default. `concrete` is the default for type definitions, and `abstract` for other definitions.
- Function definitions can be written in the normal Haskell style with type signatures and pattern matching. E.g.,

  ```
  last :: (a::#) |-> List a -> a =
      \ (a :: #) |-> \ (l::List a) ->
      case l of
      (x : (Nil)) -> x
      (x : xs) -> last xs
  ```

  can be written

  ```
  last :: (a::#) |-> List a -> a
  last (x : (Nil)) = x
  last (x : xs) = last xs
  ```

- If a definition is preceded by the keyword `type` it is assumed to have type `#` and all its arguments have default type `#`. E.g.
  `P :: # -> # = \ (a :: #) -> a->Bool`
  can be written
  `type P a = a->Bool`
- A data type definition can be written in the same way as in Haskell. This corresponds to several bindings. First one for the type itself, then one for each constructor in the type. E.g. the definition
  `data Maybe a = Nothing | Just a`
  corresponds to the definitions

  ```
  Maybe :: # |-> # =
    \(a::#) |-> data Nothing | Just a
  Nothing :: (a::#) |-> Maybe a =
    \(a::#) |-> Nothing@(Maybe a)
  Just :: (a::#) |-> a -> Maybe a =
    \(a::#) |-> \(x::a) -> Just@(Maybe a) x
  ```

- Cayenne has a `let` expression that is like the Haskell `let` expression. This can be translated into a record expression.

- To make access to record components more convenient there is an `open` expression that "opens" a record and makes its components available. The `open` construct explicitly names the components that should be visible. E.g. "`open movePoint d p use x, y in dist x y`". The "`open`" expression can easily be translated to a "`let`" expression.
- A value of record type can be coerced to a value of a different record type if the result type is the same as the original except that it has fewer fields. The coercion is written "*expr* `::` *type*" and translates to a let expression.
- Type signatures can be omitted in many places. Even if the basic rule is that all Cayenne definitions should have a type signature it is easy to relax this rule somewhat. With the relaxed rule Cayenne programs have about the same number of type signatures as the corresponding Haskell program would have and they place no big burden on the programmer.
- $\#_1$ can be written as `#`.
- A Haskell-like "`do`" notation can be used for monads.
- The Haskell layout rule is used to avoid braces and semicolons. The keywords `case`, `do`, `let`, `sig`, and `struct` triggers it.

3.3 Modules

Cayenne does not really have any modules in the traditional sense, all it has is named expressions that exist in a global name space. Module names are distiguished by having a "`$`" in their names. The module name space can be viewed as hierarchical with "`$`" as the name separator (like how UNIX path names use "`/`" or how Java names use "`.`"). Module identifiers can be used freely in expressions without any explicit import declaration (just as in Java).

A module definition looks like a simple definition except that it is preceeded by the keyword `module`. The type in the definition is not necessary and it can be left out. A module can also have `concrete` visibility. This plays the same role here as it does for records, i.e., you can make the value of a module known instead of only its type.

Some sample modules:

```
module foo$bar = struct
data Nat = Zero | Succ Nat

module foo$baz =
open System$Int use Int, (+) in
struct
inc :: Int -> Int
inc x = x+1
dec :: Int -> Int
dec x = x-1
```

Modules are the units of separate compilation. To compile a module, only the types of the modules it refers to need be known.

4 The Cayenne type system

4.1 Translucent sums

Many Haskell modules export types in a non-abstract way, i.e., the type is exported so that not only the name of the type, but also its constructors are known. E.g.

```
module Tree(Tree(..), depth) where
data Tree a = Leaf | Node (Tree a) a (Tree a)
depth :: Tree a -> Int
depth Leaf = 0
depth (Node l _ r) = 1 + (depth l `max` depth r)
```

If we try to write the corresponding Cayenne record we get

```
module ex$Tree = struct
data Tree a = Leaf | Node (Tree a) a (Tree a)
depth :: (a :: #) |-> Tree a -> Int
depth (Leaf) = 0
depth (Node l _ r) = 1 + (depth l `max` depth r)
```

which has type

```
sig
  Tree :: # -> #
  depth :: (a :: #) |-> Tree a -> Int
```

This is definitely not what we had in mind, because from this signature we can only see that `Tree` is a type constructor, but we cannot see its definition. We could try and remedy this by saying that to use a module, not only must its signature be known, but its actual value as well. This way, we would have the definition of `Tree` available. But this is also not what we intended, because this would reveal the definition of `depth`, which we may not want to reveal to users of the `ex$Tree` module.

For this reason we introduce the possibility for each record component to specify if it should be fully known or only known with its type. We then write[9]

```
module ex$Tree = struct
concrete
data Tree a = Leaf | Node (Tree a) a (Tree a)
abstract
depth :: (a :: #) |-> Tree -> Int
depth (Leaf) = 0
depth (Node l _ r) = 1 + (depth l `max` depth r)
```

[9] The `abstract` and `concrete` keywords are actually superfluous in this example because the default visibility is the same as those indicated by the keywords.

which has type

```
sig
  Tree :: # -> # =
     data Leaf | Node (Tree a) a (Tree a)
  Leaf :: (a::#) |-> Tree a =
     \ (a::#) |-> Leaf@(Tree a)
  Node :: (a::#) |-> Tree a -> a -> Tree a
          -> Tree a =
     \ (a::#) |-> \ (l::Tree a) -> \ (x::a) ->
     \ (r::Tree a) -> Node@(Tree a) l x y
  depth :: (a :: #) |-> Tree a -> Int
```

This is a very peculiar type because it not only specifies the types of the `Tree`, `Leaf`, and `Node` components, but also their exact values. Any record of this type will have a `Tree` etc. with exactly these values, whereas the value of `depth` may differ.

This idea comes from the type system for the SML module system where these types are called translucent sums, [Lil97], or the similar notion of singleton kinds. A similar construct is also present in Cardelli's Quest, where it is called manifest definitions, [Car94].

4.2 Typing and evaluation rules

The Cayenne typing rules are given in table 1 and table 2.

Some simplifications have been made to the typing rules for the purpose of presentation. In data type each constructor has exactly one argument which must be of value type. Furthermore, the order of the definitions in a `struct`/`sig` does not matter in real Cayenne, where as they do in the rules.

The stratification showed in the typing rules is also a simplification of the one used in Cayenne. The type of types as used in the rules is $\#_n$, but in actuality it is $\#_{n,m}$. The first subscript is derived as shown in the typing rules and the second we get by replacing min by max in the Prod rule. The reason for two subscripts is that the first number is necessary for getting the type erasure to be possible, and the second one is necessary if we want the logic to be sound (if recursion is removed).

The environment (or assumtions), Γ, in the typing rules may contain variables with their types, as is usually the case. But they may also contain variables with their types as well as their *values*. The reason for the values is that we sometimes need values to enable reductions during type checking. It is the Rec rule that introduces values into Γ.

The fact that Cayenne has dependent types shows up in a few places in the typing rules. In the App rule the term $f\ a$ has a type that may depend on x, so x is replaced with the actual value in B. Furthermore, in the Case rule in each arm the type of the arm may depend of the scrutinized variable so a substitution is performed here as well.

The translucent sums show up in the SelE rule where a term $e.l$ can be reduced even if only the type of e is known. This reduction is only performed during type checking and never during normal reduction (execution).

Because of a lack of time, we have not yet proved essential theorems about the Cayenne type system, such as soundness and the subject reduction theorem. While we believe them to be true, and they have been proved similar systems, they have not proved for a system with dependent types and translucent sums.

The Cayenne evaluation rules, table 4, are unsurprising. Note that because definitions in a `struct` are recursive some care has to be exercised.

A problem with substitution Substitution in Core Cayenne as described by the rules in this section suffers from a problem: it does not really work; there are some unavoidable name clashes. We illustrate the problem with an example. To make the example shorter we omit types and use a `let` expression which could be translated to a record expression.

```
struct
    x = 1
    z = let y = x
        in  struct x = y
```

As we can easily see the `z` component of this record is a record with an `x` component with value `1`. Let us apply the standard reduction rule for `let`, namely $\texttt{let}\ x = e\ \texttt{in}\ e' \longmapsto e'[x \mapsto e]$.

```
struct
    x = 1
    z = struct x = x
```

This is clearly not the same value as we had before; the `x` has been captured when it should not be. Note that we cannot rename either of the two `x`s since the name of the labels appear in the type; renaming them would change the type.

This problem is annoying, but can be handled easily. All we need to do is to have two different names for all labels. One name is the label itself as it appears in the type and the other name is the name that is bound inside the record. The second name is not part of the type and can clearly be α-converted when necessary. To avoid cluttering the typing rules even more we will not introduce any notation for this in the rules, instead we assume that the problem can be handled if needed. A similar solution is used in [Bet98].

4.3 Type checking

Type checking of Cayenne is basically simple, just because it is type checking rather than type deductions, like e.g. Haskell uses. Type checking proceeds in a single traversal of the syntax tree. On the way down the environment (Γ) is extended with the types (and sometimes values) of bound identifiers. Since Cayenne has explicit types the type of each bound identifier is known. On the

$$\frac{}{\Gamma \vdash \#_n \in \#_{n+1}}\ \textsf{Star}$$

$$\frac{\Gamma \vdash A \in s}{\Gamma, x \in A \vdash x \in A}\ \textsf{Var}$$

$$\frac{\Gamma \vdash A \in s \quad \Gamma, x \in A \vdash B \in t}{\Gamma \vdash (x::A)\texttt{->}B \in t}\ \textsf{Pi}$$

$$\frac{\Gamma, x \in A \vdash b \in B \quad \Gamma \vdash (x::A)\texttt{->}B \in t}{\Gamma \vdash \backslash(x::A)\texttt{->}b \in (x::A)\texttt{->}B}\ \textsf{Lam}$$

$$\frac{\Gamma \vdash f \in (x::A)\texttt{->}B \quad \Gamma \vdash a \in A}{\Gamma \vdash f\ a \in B[x \mapsto a]}\ \textsf{App}$$

$$\frac{\Gamma \vdash A_1 \in \#_1 \quad \ldots \quad \Gamma \vdash A_n \in \#_1}{\Gamma \vdash \texttt{data}\ C_1\ A_1 \mid \ldots \mid C_n\ A_n \in \#_1}\ \textsf{Data}$$

$$\frac{\Gamma \vdash T \in \#_1}{\Gamma \vdash C_k \texttt{@} T \in A_k\texttt{->}T}\ \textsf{Con}$$
where $T \equiv \texttt{data}\ C_1\ A_1 \mid \ldots \mid C_n\ A_n$

$$\frac{\begin{array}{r}\Gamma \vdash x \in \texttt{data}\ C_1\ A_1 \mid \ldots \mid C_n\ A_n \\ \Gamma, x_1 \in A_1 \vdash e_1 \in A[x \mapsto C_1\ x_1] \\ \vdots \\ \Gamma, x_n \in A_n \vdash e_n \in A[x \mapsto C_n\ x_n]\end{array}}{\Gamma \vdash \texttt{case}\ x\ \texttt{of}\ \{\ C_1\ x_1\texttt{->}e_1\texttt{;}\ldots\texttt{;}C_n\ x_n\texttt{->}e_n\texttt{\}::A} \in A}\ \textsf{Case}$$

$$\frac{\begin{array}{r}\Gamma \vdash A_1 \in \#_{u_1} \\ \Gamma,\ l_1 \in A_1 \vdash A_2 \in \#_{u_2} \\ \vdots \\ \Gamma,\ l_1 \in A_1, \cdots, l_{n-1} \in A_{n-1} \vdash A_n \in \#_{u_n} \\ \vdots \\ \Gamma,\ l_1 \in A_1, \cdots, l_n \in A_n \vdash e_j \in A_j \\ \vdots\end{array}}{\Gamma \vdash \texttt{sig\{}\ l_1\texttt{::}\gamma_1\texttt{;}\ldots l_n\texttt{::}\gamma_n\texttt{\}} \in \#_{min\{u_i\}}}\ \textsf{Prod}$$
where each γ_i is either A_i or "$A_i = e_j$"

$$\frac{\begin{array}{r}\Gamma \vdash A_1 \in s_1 \\ \Gamma, \Delta \vdash e_1 \in A_1 \\ \Gamma,\ l_1 \in A_1 \vdash A_2 \in s_2 \\ \Gamma, \Delta \vdash e_2 \in A_2 \\ \vdots \\ \Gamma,\ l_1 \in A_1, \cdots, l_{n-1} \in A_{n-1} \vdash A_n \in s_n \\ \Gamma, \Delta \vdash e_n \in A_n\end{array}}{\Gamma \vdash \begin{array}{c}\texttt{struct\{}\ p_1\ a_1\ l_1\texttt{::}A_1\texttt{=}e_1\texttt{;}\ldots p_n\ a_n\ l_n\texttt{::}A_n\texttt{=}e_n\texttt{\}} \\ \in \texttt{sig\{}\ \ldots l_i\texttt{::}\gamma_i\texttt{;}\ldots\texttt{\}}\end{array}}\ \textsf{Rec}$$
where l_i is present iff $p_i = \texttt{public}$,
γ_i is "$A_i = e_i$" if $a_i = \texttt{concrete}$ otherwise A_i
$\Delta \equiv l_1 \in A_1 = e_1, \cdots, l_n \in A_n = e_n$

$$\frac{\Gamma \vdash e \in \texttt{sig\{}\ \ldots l_i\texttt{::}\gamma_i\texttt{;}\ldots\texttt{\}}}{\Gamma \vdash e.l_i \in A_i[\ldots, l_k \mapsto e.l_k, \ldots]}\ \textsf{Sel}$$

Table 1. Core Cayenne typing rules

$$\frac{\Gamma \vdash a \in A \quad \Gamma \vdash B \in s \quad \Gamma \vdash A \approx B}{\Gamma \vdash a \in B}\ \mathsf{Conv}$$

$$\frac{\Gamma \vdash A \in s \quad \Gamma \vdash \delta}{\Gamma, x \in A \vdash \delta}\ \mathsf{Weak}$$

$$\frac{\Gamma \vdash a \in A \quad \Gamma, x \in A \vdash \delta}{\Gamma, x \in A = a \vdash \delta}\ \mathsf{WeakE}$$

Table 2. Core Cayenne typing rules, continued

$$\frac{\Gamma \vdash a \in A \quad \Gamma \vdash b \in A \quad \Gamma \vdash a \approx b}{\Gamma \vdash C[a] \approx C[b]}\ \mathsf{Congr}$$

where $C[]$ is any context

$$\frac{\Gamma \vdash a \in A \quad a \longmapsto b}{\Gamma \vdash a \approx b}\ \mathsf{Red}$$

$$\frac{}{\Gamma, x \in A = e \vdash \Gamma \vdash x \approx e}\ \mathsf{Lookup}$$

$$\frac{\Gamma \vdash e \in \texttt{sig\{}\ \ldots l_i :: A_i = e_i \texttt{;} \ldots \texttt{\}}}{\Gamma, l_1 \in A_1, \cdots, l_n \in A_n \vdash e.l_i \approx e_i}\ \mathsf{SelE}$$

Table 3. Core Cayenne equality rules

$$(\backslash x :: t \texttt{->} f)e \longmapsto f[x \mapsto e]$$
$$e.l_k \longmapsto e_k[\ldots, l_k \mapsto e.l_k, \ldots]$$
where $e \equiv \texttt{struct\{}\ldots \texttt{public}\ a_k\ l_k :: t_k \texttt{=} e_k \texttt{;} \ldots \texttt{\}}$
$$\texttt{case}\ C_k \texttt{@}t\ e\ \texttt{of}\ \ldots\ C_k\ x_k \texttt{->}\ e_k \texttt{;} \ldots \longmapsto e_k[x_k \mapsto e]$$

Table 4. Core Cayenne evaluation rules

$$(\backslash (x :: t) \texttt{->} f)^* \to \backslash x \texttt{->} f^*, \text{ if } t \in \#_1$$
$$(\backslash (x :: t) \texttt{->} f)^* \to f^*, \text{ if } t \notin \#_1$$
$$(f\ e)^* \to f^*\ e^*, \text{ if } e \in t \text{ and } t \in \#_1$$
$$(f\ e)^* \to f^*, \text{ if } e \in t \text{ and } t \notin \#_1$$
$$((x :: t) \texttt{->} f)^* \to \bullet$$
$$\texttt{struct\{}\ l_1 :: t_1 \texttt{=} e_1 \texttt{;} \ldots\ l_n :: t_n \texttt{=} e_n \texttt{\}}^* \to \texttt{struct\{}\ldots\ l_k \texttt{=} e_k^* \ldots \texttt{\}}, \text{ for those } l_k \text{ where } t_k \in \#_1$$
$$(e.l)^* \to e^*.l$$
$$\texttt{sig\{}\ \ldots\ \texttt{\}}^* \to \bullet$$
$$(C \texttt{@}t)^* \to C$$
$$(\texttt{case}\ e\ \texttt{of}\ C_1\ x_1 \texttt{->}\ e_1 \texttt{;} \ldots\ C_n\ x_n \texttt{->}\ e_n)^* \to \texttt{case}\ e^*\ \texttt{of}\ C_1 \texttt{->}\ e_1^* \texttt{;} \ldots\ C_n \texttt{->}\ e_n^*$$
$$\texttt{data} \ldots\ ^* \to \bullet$$
$$\#_n^* \to \bullet$$
$$x^* \to x, \text{ if } x \in t, t \in \#_1$$
$$x^* \to \bullet, \text{ if } x \in t, t \notin \#_1$$

Table 5. Type erasure transformation

$$(\backslash x \texttt{->} f)\, e \rightharpoonup f[x \mapsto e]$$
$$e.l_k \rightharpoonup e_k[\ldots, l_k \mapsto e.l_k, \ldots]$$
$$\text{where } e \equiv \texttt{struct\{}\ldots\ a_k\ l_k\texttt{=}e_k\texttt{;}\ldots\texttt{\}}$$
$$\texttt{case}\ C_k\ e\ \texttt{of}\ \ldots\ C_k\ x_k\texttt{->}\ e_k\texttt{;}\ldots \rightharpoonup e_k[x_k \mapsto e]$$

Table 6. Core Cayenne typeless evaluation rules

way up the type of each subexpression can the be computed and checked. A complication arises when a typing rules has more than one occurence of a type, like A in the App rule in table 1. For these cases we need to check if the two types derived from the bottom up derivation are the same, and if they are not identical the Conv rule can be used to make them equal (assuming the program is type correct). For a strongly normalizing language without translucent types the Conv rule is uses $=_\beta$ for $\approx$. This relation is easy to implement; just compute the normal forms of the two types and compare those. Since Cayenne is not strongly normalizing this is not an option. The equivalence of two arbitrary expression is undeciable. For this reason, we can not implement anything but an approximation of the Conv rule and the equality rules (table 3). This is a tricky part of the Cayenne type checker since if the equivalence test is implemented in a naïve way type checking can easily loop.

4.4 Undecidability in practice

So type checking Cayenne is undecidable. This is unfortunate, but unavoidable for a language like Cayenne. How bad is it in practice to have an undecidable type checker? This question can only be answered by practical experiments. The Cayenne programs we have tried to date range from ordinary Haskell style programs, to programs using dependent types, to proofs of mathematical propositions. The total size of these programs are only a few thousand lines, but so far the experience shows that it works remarkably well.

Having undecidable type checking means that the type checker might loop. This is clearly not a user friendly type checker. So instead the implemented type checker has an upper bound on the number of reduction steps that it may perform. If this limit is exceeded the type checker will report this. Most of the type errors from the Cayenne compiler are similar to those that any other language would give. Very infrequently does the type checker report that it did not terminate within the prescribed number of steps. Most often, this is the result of a type error, but sometimes the type expression is just too complicated and the number of reduction steps must be increased (the number of reduction steps is a compiler flag).

The type checker can thus give one of three answers: type correct (meaning that the program will not go wrong when run), type incorrect, or "don't know"[10].

[10] On a real machine Hindley-Milner type checking has the same problem, but the third alternative is usually spelled "Out of memory" instead.

There are other languages with undecidable type checking, e.g., Quest [Car94] (which has a type system based on $F_{\omega<}$) and Gofer [Jon94], but it is usually more difficult to make these systems loop.

5 Cayenne as a proof system

Since Cayenne has unrestricted recursion, this means that every type is inhabited by at least one element, namely $\bot$. Thus, proofs made in Cayenne cannot really be trusted as proofs, since any proposition can be proved by $\bot$. If proper checking is done, it is often[11] possible to ensure that a proof is valid, but no such checking is done at the moment.

Even if a proofs expressed in Cayenne cannot be trusted because they pass the type checker it is still valuable to have the encoding of predicate logic in the language. Firstly, it allows us to express properties about programs within the language even if we provide no proofs at all. It is better to have this ability within the language than to use comments or leave out those properties completely. Secondly, even if a proof cannot be trusted one can argue that a proof that has been checked, but may be $\bot$, is better than a proof that is not checked at all.

6 Implementation

Implementing Cayenne is fairly straight forward; it is like any other functional language. One decision that has to be made is what to do with types at runtime.

6.1 Erasing types

Cayenne treats types like first class values. Does that mean that the types have to be present at run time, passed around as arguments, stored in data structures, etc? No, they do not. There is no language construct, e.g., `casetype`, that allows a ground value — which is all that can be observed in a program — to depend on a type. Hence, types do not have to be present at run time. Erasing types consists of removing all arguments and record components that have type $\#_n$ or are functions computing something of type $\#_n$. In [Car88] it is claimed that type erasing is not possible and that the distinction between compile-time and run-time is blurred with dependent types. We claim that this is not the case with the variant of dependent types used in Cayenne.

What we need to show is that evaluating an expression with types erased yields the same result as evaluating it with the types left in.

Definition An expression, e, has *value type* if $e \in t$ and $t \in \#_1$

Theorem If e has value type and $e \longmapsto^* v$ then $e^* \rightharpoonup^* v^*$.

We first prove a useful lemma.

Lemma If e has value type, then e^* contains no $\bullet$.

[11] Not always, of course, since then we would have to solve the halting problem.

Proof We assume that the expression to transform is of value type, and show that each invocation of the transformation on a subexpression is also on an expression of value type.

Cases A λ-expression $\backslash(x::t)\texttt{->}f$ has type $(x::t)\texttt{->}r$, where r is the type of f. According to the assumption $(x::t)\texttt{->}r$ has type $\#_1$ and typing rule Pi shows that then r has type $\#_1$ as well. Thus the transformation of f is also on an expression of value type.

For an application $f\ e$, according to the definition of *, the transformation is only applied to e if it is of value type. f has type $(x::t)\texttt{->}r$ and $f\ e$ has type r, if r has type $\#_1$ then, again according to typing rule Pi, $(x::t)\texttt{->}r$ has type $\#_1$, so the transformation of f fulfills the assumption.

The transformation cannot be applied to a function type since this does not have value type.

For a record value `struct{ ... }` the transformation is only applied to subexpressions of value type according to the definition of *.

If a record selection $e.l$ is of value type then the field l must be of value type. If one field of a record type type has type $\#_1$ then the whole record type has type $\#_1$ according to typing rule Prod (which takes the *min* of all the types), so the subexpression e (of $e.l$) must have value type.

The transformation cannot be applied to a `sig{ ... }` value since it is not of value type.

The lemma is trivially true for a contructor expression.

For a `case` expression the transformation is applied to the scrutinized expression, which is always of value type (typing rule Data) and to all the right hand sides. The right hand sides are of value type if the whole case expressions is.

The transformation cannot be applied to a `data` value since it is not of value type.

The transformation cannot be applied $\#_1$ since it is not of value type.

The lemma is true for variables according to the definition of *.

Corollary A transformed expression of value type contains no variables that were not of value type in the original expression.

Proof Variables that are not of value types are translated to $\bullet$, but there are no $\bullet$ in the expression, hence there can be such variables.

Lemma The substitution lemma states that type erasure commutes with substittion: $(e[x \mapsto t])^* = e^*[x \mapsto t^*]$.

Proof By structural induction over the expression syntax.

We can now return to proving the type erasure theorem. First we prove that if e has value type and $e \longmapsto f$ then $e^* \rightharpoonup f^*$ or $e^* = f^*$. The theorem then follows simply by induction on the length of the reduction sequence.

The single step version of the theorem is proved by case analysis on the three different (typed) reduction kinds.

Cases If the reduction is $(\backslash x::t\texttt{->}f)e \longmapsto f[x \mapsto e]$ then the translation of the redex is either $(\backslash x\texttt{->}f^*)e^*$ in which case there is a corresponding untyped reduction step (according to the substitution lemma). Or the translation of the

redex is f (if x and e do not have value type). In this case $f^* = f^*[x \mapsto e^*]*$ since x does not occur in f^* (according to the corollary).

If the reduction is a selection the selected label could either be left in the transformed `struct` or it could have been erased. But since the expression $e.l$ has value type this means that the label has value type and it must thus be left in the `struct`. There is then an exactly corresponding untyped reduction.

If the reduction is a case reduction there is an exactly corresponding untyped reduction.

QED

6.2 Keeping types

By keeping types at runtime it is possible to do computations on types and base control decision on the dynamic type of values. With runtime types we could have a `casetype` language construct. Keeping types around at runtime have some advantages, like mostly tag-free garbage collection, as used in TIL, [TMC+96,Mor95].

6.3 The current implementation

The current implementation of Cayenne is written in Haskell and translates Cayenne to untyped LML. The compiler consists of about 5500 lines, a third of which is the actual type checker. The compiler parses Cayenne, does type checking and various other checks, erases types and then translates the resulting code into LML. The LML code is then compiled with the LML compiler, [AJ89], with type checking turned off. This works because the LML compiler does not rely on a the fact that the program is type correct in the Hindley-Milner type system; all the compiler assumes is that the program "makes sense".

A snapshot of the current implementation can be found on the Web at `http://www.cs.chalmers.se/~augustss/cayenne/`.

7 Related work

There are many logical frameworks (proof checking systems) that are based on dependent types. Some examples, among many, are ALF [MN94,Nor93,ACN90], CoC [CH86,CH88], ELF [Pfe89,Fra91,HHP93], Lego [Pol94], and NuPRL [Con86]. All these systems are primarily designed for making (constructive) proofs even if many of them can also execute the resulting proofs or extract a program from them. Our approach is different in that we want to make a programming language, not a proof system, but of course there are big similarities.

There are few programming languages with dependent types. Cardelli's Quest, [Car94], have similarities with Cayenne, but the final version of Quest does not have the full dependency where types can depend on values. Russell, [BDD89], has dependent types, but the notion of type equality is "name equality" rather than the "structural equality" of Cayenne. Russell does not do full evaluation

during type checking so it would not be able to do, e.g., the `printf` example. Russell also has a different notion of what a type is.

8 Future work

There are many ways to continue the work on Cayenne and related languages. First, and foremost, is to gain more experience with a language with dependent types, both to see how dependent types can be used and to see how undecidable type checking works out.

Another interesting line of work is to make a partial evaluator for this kind of language. Since types and values are combined, a partial evaluator would serve both as a type specializer (as used in, e.g., [Aug93,PJ93]) and a traditional partial evaluator.

To make the record types more useful, subtyping could be added. Subtyping in the presence of dependent types has been studied in [Bet98].

As a proof of concept the Cayenne compiler should, of course, be rewritten in Cayenne.

9 Acknowledgments

A big thanks to Jessica for improving my English. The programming logic group at Chalmers has over the years provided me with enough background material to finally try to make a programming language with dependent types. A special thanks to Theirry Coquand for fruitful discussions and examples of how to write type checkers for dependent types. Thomas Johnsson, Niklas Röjemo and Dan Synek provided me with feedback on this paper as did the anonymous ICFP referees.

References

[ACN90] L. Augustsson, T. Coquand, and B. Nordström. A short description of Another Logical Framework. In *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 39–42, 1990.

[AJ89] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.

[Aug93] Lennart Augustsson. Implementing Haskell Overloading. In *Proc. 6th Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 65–73. ACM Press, June 1993.

[BDD89] H. Boehm, A. Demers, and J. Donahue. A Programmer's Introduction to Russell. Technical report, Cornell University, 1989.

[Bet98] Gustavo Betarte. *Dependent Record Types and Algebraic Structures in Type Theory.* PhD thesis, Department of Computing Science, University of Göteborg, Göteborg, Sweden, February 1998.

[Car88] Luca Cardelli. Phase Distinction in Type Theory. Research report, DEC SRC, 1988.

[Car94] Luca Cardelli. The Quest Language and System. Research report, DEC SRC, 1994.

[CH86] Thierry Coquand and Gérard Huet. The Calculus of Constructions. Technical Report 530, INRIA, Centre de Rocquencourt, 1986.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[Con86] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Dan98] Olivier Danvy. Formatting Strings in ML. Technical Report RS-98-5, BRICS, Department of Computer SCience, University of Aarhus, Denmark, March 1998.

[Fra91] Logical Frameworks. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *LICS'89*, pages 149–181. Cambridge University Press, 1991.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *JACM*, 40(1):143–184, 1993.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[Jon94] Mark P. Jones. The implementation of the Gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Department of Computer Science, Yale University, New Haven, Connecticut, USA, May 1994, May 94.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997. CMU-CS-97-122.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical report, LFCS Technical Report ECS-LFCS-92-211, 1992.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[Mor95] Greg Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, 1995.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Nor93] Bengt Nordström. The ALF proof editor. In *Proceedings 1993 Informal Proceedings of the Nijmegen workhop on Types for Proofs and Programs*, 1993.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *LICS'89*, pages 313–322. IEEE, June 1989.

[PJ93] John Peterson and Mark P. Jones. Implementing Type Classes. In *Proceedings of ACM SIGPLAN Symposium on Programming Language Design and Implementation*, June 1993.

[Pol94] Robert Pollack. *The Theory of Lego A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[TMC+96] David Tarditi, Greg Morrisett, Pery Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A Type-directed Optimizing Compiler for ML. Technical Report CMU-CS-96-108, School of Computer Science, Carnegie Mellon University, February 1996.

A The Eq class

```
module example$Eq =
#include Prelude
struct

data Absurd =

data Truth = truth

absurd :: (a :: #) |-> Absurd -> a
absurd i = case i of { }

type (<=>) a b = sig { impR :: a->b; impL :: b->a; }

concrete
Lift :: Bool -> #
Lift (False) = Absurd
Lift (True)  = Truth

concrete
LiftBin :: (a:: #) |-> (a -> a -> Bool) -> Rel a
LiftBin |a op = \(x::a) -> \(y::a) -> Lift (op x y)

type Rel a = a -> a -> #

concrete
Refl :: (a :: #) |-> Rel a -> #
Refl |a R = (x::a) -> x `R` x

concrete
Symm :: (a :: #) |-> Rel a -> #
Symm |a R = (x,y::a) -> x `R` y -> y `R` x

concrete
Trans :: (a :: #) |-> Rel a -> #
Trans |a R = (x,y,z::a) -> x `R` y -> y `R` z -> x `R` z

concrete
Equiv :: (a :: #) |-> Rel a -> #
Equiv R = sig
  refl :: Refl R
  symm :: Symm R
  trans :: Trans R

--------
-- The Eq "class", with equivalence proof
type Eq a = sig
  (==) :: a -> a -> Bool
  equiv :: Equiv (LiftBin (==))

--------
-- Equality on Unit
Eq_Unit :: Eq Unit
Eq_Unit = struct
```

```
  (==) (unit) (unit) = True

  equiv = struct
    refl (unit) = truth
    symm (unit) (unit) p = p
    trans (unit) (unit) (unit) p q = p

--------
-- Equality on Bool
Eq_Bool :: Eq Bool
Eq_Bool = struct
  (==) (False) (False) = True
  (==) (True)  (True)  = True
  (==) _       _       = False

  equiv = struct
    refl (False) = truth
    refl (True)  = truth
    symm (False) (False) p = p
    symm (False) (True)  p = absurd p
    symm (True)  (False) p = absurd p
    symm (True)  (True)  p = p
    trans (False) (False) (False) p q = q
    trans (False) (False) (True)  p q = absurd q
    trans (False) (True)  _     p q = absurd p
    trans (True)  (False) _       p q = absurd p
    trans (True)  (True)  (False) p q = absurd q
    trans (True)  (True)  (True)  p q = q

--------

private
liftAndL :: (x,y::Bool) ->
            Lift (x && y) -> Pair (Lift x) (Lift y)
liftAndL (False) _       a = absurd a
liftAndL (True)  (False) a = absurd a
liftAndL (True)  (True)  t = (t, t)

private
liftAndR :: (x,y::Bool) ->
            Pair (Lift x) (Lift y) -> Lift (x && y)
liftAndR (False) _       (a, _) = a
liftAndR (True)  (False) (_, a) = a
liftAndR (True)  (True)  (t, _) = t

private
isoEquiv :: (a :: #) |->
     (p, q :: Rel a) -> ((x, y :: a) ->
     p x y <=> q x y) -> Equiv p -> Equiv q
isoEquiv p q iso eqp = struct
  refl x = (iso x x).impR (eqp.refl x)
  symm x y lp =
    (iso y x).impR (eqp.symm x y ((iso x y).impL lp))
  trans x y z lp lq = (iso x z).impR
    (eqp.trans x y z ((iso x y).impL lp) ((iso y z).impL lq))
```

```
-- Equality on pairs.
Eq_Pair :: (a,b :: #) |-> Eq a -> Eq b -> Eq (Pair a b)
Eq_Pair eqa eqb = struct
  (==) (x, x') (y, y') = eqa.(==) x y && eqb.(==) x' y'

  private
  eq :: Pair a b -> Pair a b -> #
  eq (x, x') (y, y') =
    Pair (LiftBin eqa.(==) x y) (LiftBin eqb.(==) x' y')

  private
  eqEq :: (x,y::Pair a b) -> eq x y <=> Lift (x == y)
  eqEq (x, x') (y, y') = struct
    impR p = liftAndR (eqa.(==) x y) (eqb.(==) x' y') p
    impL p = liftAndL (eqa.(==) x y) (eqb.(==) x' y') p

  private
  equivEq :: Equiv eq
  equivEq = struct
    refl (x, x') = (eqa.equiv.refl x, eqb.equiv.refl x')
    symm (x, x') (y, y') (pxy, pxy') =
      (eqa.equiv.symm x y pxy, eqb.equiv.symm x' y' pxy')
    trans (x, x') (y, y') (z, z') (pxy, pxy') (pyz, pyz') =
      (eqa.equiv.trans x y z pxy pyz,
       eqb.equiv.trans x' y' z' pxy' pyz')

  equiv = isoEquiv eq (LiftBin (==)) eqEq equivEq
```

B The tautology function

```
module example$taut =
#include Prelude
struct
data Nat = Zero | Succ Nat

concrete
TautArg :: Nat -> #
TautArg (Zero)   = Bool
TautArg (Succ m) = Bool->TautArg m

taut :: (n::Nat) -> TautArg n -> Bool
taut (Zero)   x = x
taut (Succ m) x = taut m (x True) && taut m (x False)


module example$tauttest =
#include Prelude
open example$taut use Nat, Zero, Succ, taut, TautArg in

let id :: Bool -> Bool
    id x = x

    implies :: Bool -> Bool -> Bool
```

```
    implies x y = not x || y

    equ :: Bool -> Bool -> Bool
    equ x y = implies x y || implies y x
in  do Monad_IO
       putStrLn (System$Bool.show (taut Zero True))
       putStrLn (System$Bool.show (taut (Succ Zero) id))
       putStrLn (System$Bool.show (taut (Succ (Succ Zero)) equ))
```