The Aldor-- language

Simon Thompson and Leonid Timochouk
Computing Laboratory, University of Kent
Canterbury, CT2 7NF, UK
{s.j.thompson,l.a.timochouk}@ukc.ac.uk

Abstract

This paper introduces the Aldor-- language, which is a functional programming language with dependent types and a powerful, type-based, overloading mechanism. The language is built on a subset of Aldor, the 'library compiler' language for the Axiom computer algebra system. Aldor-- is designed with the intention of incorporating logical reasoning into computer algebra computations.

The paper contains a formal account of the semantics and type system of Aldor--; a general discussion of overloading and how the overloading in Aldor-- fits into the general scheme; examples of logic within Aldor-- and notes on the implementation of the system.

1 Introduction

Aldor-- is a dependently-typed functional programming language, which is based on a subset of Aldor [16]. Aldor is a programming language designed to support program development for symbolic mathematics, and was used as the library compiler for the Axiom computer algebra system [8].

Aldor-- embodies an approach to integrating computer algebra and logical reasoning which rests on the formulas-as-types principle, sometimes called the Curry-Howard isomorphism [14], under which logical formulas are represented as types in a system of dependent types.

Aldor and Axiom have dependent types but, as was argued in earlier papers [11, 15], because there is no evaluation within the type-checker it is not possible to see the Aldor types as providing a faithful representation of logical assertions. A number of other implementations of dependently-typed systems already exist, including Coq [13], Alfa [1] and the Lego proof assistant [10]. The work reported here is novel in two respects.

- The logic is embedded in a system which is intended to support symbolic mathematics, rather than a system explicitly designed to aid the construction of type-theoretic proofs.
- Mathematical practice makes widespread use of *overloading*, and for this reason Aldor was designed to incorporate the overloading of identifiers in as freewheeling as possible a form. This has particular consequences for a system of dependent types in which type-checking and evaluation are tightly coupled.

This paper describes the syntax, semantics and type system of Aldor--. We use a Haskell [7] notation for the syntax, and we describe four principal semantic meta-types, also reflected in the syntax of the language.

Manifest Types. Examples of these include base types, such as Booleans, numbers and strings; function, record and union types; abstract data types and categories, and `TypeT`, the type of all types.

Manifest Values. These include values from the manifest types, and indeed these types themselves, which belong to the type `TypeT`.

Expressions. Expressions are entities which can be evaluated to yield manifest values. Manifest values may, of course, be manifest types.

Functional Blocks. These blocks represent the hierarchical definition structure of the language. They can also be thought of as closures, combining an expression with a context which gives values to free variables in that expression.

Central to the explanation of the semantics of Aldor-- is the concept of *reduction*. Reduction can be applied to objects from any of the Aldor-- meta-types. The functional semantics of Aldor-- involves β-reduction of function applications and δ-reduction of defined identifiers. Reductions can be classified as being of two sorts.

Evaluation. If an expression or a functional block can be reduced to a value, this is called (full) evaluation.

Normalisation. All other cases are known as normalisation or *partial* evaluation.

For example, if no value for `x` is available in the current context, then `x+3` is normalised to itself. If, on the other hand, `x` is bound to `4`, then `x+3` evaluates to `7`.

The remainder of the paper is as follows. We first present a discussion of overloading in general, as a context for the overloading in Aldor--; this is followed by an introduction to the syntax of Aldor-- and then by an overview of overloading in Aldor--. Section 5 contains a formal account of reduction in the language, and Section 6 formally describes the type system of Aldor--. Section 7 contains an overview of the implementation of the system, and Section 8 covers a number of illustrative examples.

2 Overloading: a survey

This section surveys the topic of overloading in programming languages in order to give a context to our later discussion of the central rôle that overloading plays in Aldor--.

An identifier is overloaded (within a particular scope) if it denotes more than one object simultaneously.

Overloading is common in mathematics, where the same multiplication symbol might be used for multiplication of two scalars, multiplication of a two vectors and multiplication of a vector by a scalar. In each case here the symbol is used with the same ‘arity’, as it denotes a binary operation in each case.

Overloaded symbols need not have the same arity: the minus sign, ‘-’ is commonly used both for unary minus, which changes the sign of a number (or indeed a vector), and also for binary subtraction.

Overloading is not confined to operators. Function names, and indeed other names, can also be overloaded. For instance, it is common in object-oriented languages to overload the constructors of a class, which are given the same name as the class itself. A constructor with no arguments might set default values for attributes, whereas explicit parameters can be used to give attributes particular values on object creation.

Why overloading?

What is the rationale for overloading a name? Two separate justifications emerge.

The first is that despite the fact that modern programming languages offer an effectively unlimited collection of possible names, the *choice of names* – especially if they are to have mnemonic value – is substantially smaller. It makes perfect sense that the size of an arbitrary data structure should be returned by `size`, for instance. Without the symbols afforded by full Unicode, there is a particular paucity of possible operators and constants in the ASCII character set; hence the decision that a unit element of whatever sort is to be denoted `1`.

Overloading of this kind is syntactic sugar, which makes a programmer more effective by making programs and libraries easier to read and write. It would be possible to replace all overloaded names; in the case of `size` by defining instead the functions `sizeBlah`, where `Blah` is the name of the data type in question.

The second rationale is that overloading provides a form (in fact various forms) of *genericity*: it is possible to use a function without knowing the type at which it is being applied. Continuing the `size` example, it is possible to find the total size of a list of data structures by summing the individual sizes:

```
totalSize = sum (map size)
```

Pushing the overloading one stage further we could in fact call the function that we are defining `size` too

```
size = sum (map size)
```

The code here will work irrespective of the particular data structures in the list (so long as the host programming language has a sufficiently liberal type discipline).

Mechanisms

Here we survey the various mechanisms for defining overloaded identifiers.

Finitary overloading

A program is a finite artifact, and as such can only contain a finite number of definitions. *A fortiori*, therefore, it can only contain a finite number of definitions for any particular identifier. In a momomprphic language, this is the whole story, except perhaps for built-ins like equality. In languages with polymorphism the situation is more complex.

In Haskell98 it is possible to declare identifiers whose overloaded definitions each cover a collection of types: the `size` function for lists will employ the same code to measure the length of any type of list, for instance.[1]

```
class Size where
  size :: a -> Integer

instance Size [a] where
  size []     = 0
  size (_:xs) = 1 + size xs
```

In this case the definition itself is still finitary – indeed the same code covers all lists – but it can be applied at an infinite number of different types. On the other hand, it can only be used at those data types for which there is an explicit instance declaration, and so cannot cover all possible data types.

[1]The `class` declaration introduces a *type class*, which is specified by a siganture; in other words a `class` declaration defines an interface. The instance declarations give implementations of that interface.

Infinitary overloading: varieties of generic programming

Suppose that you would like to define a function that will work over all types, returning (say) the size of its argument, measured in some way. It is clear that this cannot use the same code at every type, and so it is necessary to define the function by cases over the different types. Such a *type case* construction can be realized in a number of different ways, but in essence a type has to be passed to the function as an argument, either explicitly or implicitly, and either structurally or by name.

Types can be represented in the language by a type `Type`. If the language has a fixed repertoire of type constructors (such as tuples, lists and function spaces), then it is possible to give an exhaustive description of the types, as elements of a data type. In pseudo Haskell the tuple type `(Int -> Int,Bool,Int)` might be represented by the value

```
Tuple [ (FunSpace IntType IntType), BoolType, IntType]
```

A function such as `size` can then examine the type representation(s) of its argument(s) within its definition. In the case of this tuple, we have

```
size (Tuple [ (FunSpace IntType IntType), BoolType, IntType])
 = size (FunSpace IntType IntType) . proj 3 1 +++
   size (BoolType) . proj 3 2 +++
   size IntType . proj 3 3
   where
   f +++ g = \x -> f x + g x
```

and `proj n m` is the function which projects from an `n`-tuple to its `m`th component. This approach goes by the general name of *intensional polymorphism* [5]. Research issues in this area centre upon finding efficient implementations of such intensionality, and, in turn, its use in finding optimal implementations of language features according to the particular data representations for language primitives.

Intensional polymprphism can be extended to cover generative type definitions, that is type definitions that generate new types. With such a facility it is impossible directly to enumerate the types of the langauge: how can one predict the names given to the types and to the type constructors, for example?

On the other hand, it is possible to describe in a finitary way all the possible structural definitions: data types are given by the fixed points of functions over types built from sum, product, function space and other type constructors. A definition of a function like `size` can be given indirectly along the following lines:

- when given an element of a product type, find the `size` of the two components and add them together;

- when given an element of a sum type, then apply whichever of the summand functions is appropriate.

These mechanisms underly the so-called ‘polytypic’ languages like PolyP and Generic Haskell [6].

As the informal description of the `size` algorithm might suggest, the functions of Generic Haskell are uniform in the construction of the type. In Strafunski [9] such general behaviour can be combined with ‘ad hoc’ definitions at particular types, giving a very elegant and powerful platform for type-directed programming.

What is *not* overloading (part 1)?

Overloading, non-determinism and multiple-valued programs

It would be possible to build a programming language in which multiple definitions of an identifier *at the same type* were permitted. For example, a language might allow multiple definitions (of constants) thus:

```
bar :: Integer

bar = 37
bar = 42
```

One can view the evaluation of expressions in this language in (at least) two different ways.

1. The language is non-deterministic, and so evaluating `bar+bar` could give one of three results, `74`, `79` and `84`.

2. The language is set-valued, and so the evaluation of `bar+bar` gives the set $\{$`74,79,84`$\}$, or bag-valued, giving $\{\{$`74,79,79,84`$\}\}$.

These approaches are perfectly legitimiate, but they are outside what we understand by overloading. Under our approach, it is expected that every expression will have a *unique* value.

Resolving overloading

If an expression containing overloaded identifiers is required to have a unique value, how can this be achieved? It is not difficult to see that the problem of deciding whether an arbitrary expression has a unique value is formally undecidable (see [3], for instance, for the background to questions like this).

A sufficient condition to ensure that an expression is single-valued is that each overloaded identifier in the expression can in fact be bound to only one definition *for reasons of typing*. A simple example is given by

```
foo :: Integer
foo = 42

foo :: String
foo = "Forty two"
```

Consider some examples

```
expr1 :: Integer
expr1 = foo + 37

expr2 :: Integer
expr2 = length foo + 37

expr3 :: Integer
expr3 = if foo==foo then length foo else foo
```

In the expressions `expr1` and `expr2` it is plain that `foo` is used at a single type.

In `expr3` `foo` is used four times: the third use is at type `String` and the fourth at `Integer`. The first two occurrences could be at either type, assuming that == denotes an overloaded equality. Despite the fact that `foo==foo` would evaluate to `True` irrespective of the type of `foo`, `expr3` would be rejected on a type-based resolution of overloading, since it is impossible to assign a unique type to the first two occurrences of `foo` or to the equality symbol, ==.

Two types of ambiguity[2]

How, in general, are ambiguous references to be resolved? There are two mechanisms at work.

The first mechanism resolves overloading in a *bottom-up* manner. A function identifier is resolved by consulting the types of its *arguments*. This permits certain sorts of overloading, such as

- using the '+' symbol for addition over both integers and real numbers. In this case, '+' will have the types

  ```
  Integer -> Integer -> Integer
  Float -> Float -> Float
  ```

- Using the '-' symbol for both binary subtraction and unary negation over integers.[3] In this case, '-' will have the types

  ```
  Integer -> Integer -> Integer
  Integer -> Integer
  ```

Bottom-up resolution characterises the C++ overloading mechanism[4]; on the other hand, Ada, for instance, allows a more general form of overloading whereby the *context* is used in a *top-down* resolution. Cases in which top-down resolution is necessary include

- overloaded literals, like `1` and overloaded constants such as `foo` above. These uses can only be resolved by viewing the context in which they are used.

- In general, cases in which function uses are resolved by their return types require context. Use of a function such as

  ```
  parse :: String -> Integer
  parse :: String -> Boolean
  ```

  in `parse str` is unresolved, and the context of the application has to be examined. Resolution can require arbitrary backup through the expression tree, since the immediately enclosing operator may be overloaded to accept either an integer or a Boolean.

[2]With apologies to William Empson.

[3]The partial applications of Haskell can cause a problem here. If `f` has types `Integer -> Integer -> Integer` and `Integer -> Integer` then it is not clear what type `f 42` has: it could be either `Integer -> Integer` and `Integer`.

[4]For a discussion of the rationale behind this decision and other C++ design decisions, see [12].

What is *not* overloading (part 2)?

Parametric polymorphism is not overloading

In a language like Pascal it is possible to define a number of different types of linked list, which differ only according to the type of their elements, but which share the same structure. A Pascal programmer is forced to (re-)define a function to return the length of a list for each list type, despite the fact that essentially the same program works at each type.

Langauges such as Haskell [7], which conform to the Hindley-Milner type discipline, allow the definition of parametric types

```
data List t = Nil | Cons t (List t)
```

where `t` is a type parameter. It is then possible to define a length function which works over all types of lists

```
length :: List t -> Integer                                          (length)
length Nil         = 0
length (Cons x xs) = 1 + length xs
```

The `length` function can be applied to lists of elements of any type, but it is important to observe that the identifier `length` denotes a single object, which can be applied at any list type. In particular, there is no ambiguity about the effect of applying `length`: the code (`length`) will be used at every list type. The definition here is an example of *parametric polymorphism*: the polymorphism ('many shapes') of the function `length` are given by the fact that the type `t` is a parameter to the definition of the function.

Overloading and coercion

Closely related to overloading is coercion, that is the implicit conversion of a value in one type to another type. The canonical example of this is the conversion of integers to floating point numbers. Suppose `1` is a literal integer and `1.0` a literal float; what will be the value of the expression that follows?

```
1 + 1.0
```

If `+` is overloaded to be both an integer and a floating point operator, then `1 + 1.0` will be ill-typed, unless either

- the integer `1` is coerced to the float `1.0`, or
- the operator `+` is also overloaded to have type

  ```
  Int -> Float -> Float
  ```

To ensure full generality in the second case then `+` should also be overloaded to type `Float -> Int -> Float`, thus further proliferating definitions. It is in the light of this that coercion is often justified, but coercion adds substantial complications to an existing language.

For instance, it gives a problem of *coherence*. Suppose that we are asked for the floating point equivalent of the addition `1+3`, there are two alternatives:

- the numbers are added as integers and the resulting integer is coerced to be a float;
- the numbers are coerced to being floats, and are then added as such.

The coherence criterion requires that the two mechanisms give the same result. In many situations this will not happer: consider the addition of `N` and `1-N` where `N` is the maximum storable integer. The first mechanism will five the answer `1`; it is by no means clear that the second will do the same.

Bounded ad hoc polymorphism is not overloading

Subclassing in object-oriented programs allows the same identifier to be bound to different definitions in different subclasses. For instance, a class of two-dimensional geometrical shapes may have an `area` method which is redefined in subclasses of triangles, squares and so forth. When the `area` method is invoked on a shape `S` say, then the particular code used will depend upon the (dynamic) class to which `S` belongs; this mechanism is called dynamic despatch.

The method `area` is polymorphic; it can be applied to all subclasses of a given class (hence the 'bounded'), and it has different definitions at different subclasses (hence 'ad hoc').

Interface-based and freeform overloading

Different overloading disciplines allows overloading in different ways. Haskell98 type classes specify which identifiers can be overloaded by presenting them in signatures. These signatures are implemented by instances which insantiate the overloaded identifiers at the instance type. In a similar way, OO methods redefined within subclasses are required to adhere to the signature of the superclass method.

By contrast, other languages do not require any uniformity of this sort in the overloading of identifiers. The work presented in this paper is of this kind.

Related work

Castagna and Chen [2] introduce a system combining overloading and dependent types. Their overloading is required to be resolved bottom up, by analysing the types of arguments passed to functions. Our overloading is resolved in a more general manner.

3 Aldor-- syntax

To begin with, we present the *abstract synatx* of Aldor--. Examples of the concrete syntax are given at the end of this section.

The syntax of Aldor-- is presented as a collection of Haskell [7] `data` and `type` definitions in Figure 1.

3.1 Aldor-- programs

An Aldor-- program, `AldorProgram`, is a top-level functional block given by the Haskell type `FB`. There are four kinds of `FB`:

`SimpleFB e`: A simple functional block is given by an Aldor-- expression, `e`, of type `Expr`.

`NestedFB dt fb`: A 'nested' FB contains a definition table, `dt`, of type `DefTable`, and an inner functional block, `fb`.

`FParmPB dt`: The third kind of FB is used for technical reasons to 'wrap' a definition table as used in declarations of function, record and union types, `FunctionT`, `RecordT` and `UnionT`. This is explained in more detail below in Section 3.5.

`VoidFB`: This is a 'placeholder' functional block, used, for example, in a declaration of the type of an object not containing an associated definition body.

```
type AldorProgram = FB

data FB  =
    SimpleFB  Expr
  | NestedFB  DefTable  FB
  | FParmFB   DTE
  | VoidFB

data  DefTable =
      DefTable  [DTE]  [QName]

data  DTE = DTE  String FB Expr

data   Expr  =
    Literal      Value
  | Identifier   QName
  | AndList      [FB]
  | OrList       [FB]
  | IfThenElse   FB  FB  FB
  | FunctionCall FB  [FB]
  | RecordCtor   [FB]
  | UnionCtor    Expr  FB
  | UnionCase    FB  Expr
  | RestrictType FB  FB
  | ConvertTo    FB  FB
  | Add1         FB
  | Add2         FB  FB
  | With1        FB
  | With2        FB  FB
  | VoidExpr
```

```
type QName = [String]

data Value =
    TypeVal    Type
  | BooleanVal Bool
  | IntegerVal Integer
  | FloatVal   Double
  | StringVal  String
  | LambdaVal  Type FB
  | BuiltInOp  String  Type
                  ([Value] -> Value)
  | RecordVal  [Value]
  | UnionVal   Expr  Value
  | ADT_Val    Type  Value
  | TrivVal

data Type =
    TrivT
  | ExitT
  | TypeT
  | BooleanT
  | IntegerT
  | FloatT
  | StringT
  | FunctionT [FB]  FB
  | BuiltInT  [Type]  Type
  | RecordT   [FB]
  | UnionT    [FB]
  | ADT       FB DefTable
  | CategoryT DefTable
```

Figure 1: The Syntax of Aldor--

3.2 Definitions and definition tables

An Aldor-- definition associates with a name

- an expression which denotes the value given to that name, and
- the type of that value.

A single definition is represented by an object of the `DTE` type, such as

`DTE n fb e`

The expression `e` here evaluates to the *type* of the value given to the name `n` by the functional block `fb`. Because of the intimate relation between types and values in Aldor--, Aldor-- types are given by a subset of Aldor-- expressions, and so a type is, in general, given by an expression rather than a manifest type. Types and type-valued expressions are described in more detail in Sections 3.3 and 3.5.

A definition table is a collection of definitions, represented as a list of `DTE`s together with a list of names of imported domains. The names can, in general, be fully qualified, and are therefore represented by the type `QName = [String]`. The import semantics is defined in Section **??**. Definition tables are themselves represented by the Haskell type `DefTable`.

3.3 Expressions

The Haskell type `Expr` lies at the heart of the program structure of Aldor--. Expressions are entities which can be evaluated. The elements of the type `Expr` are as follows.

`Literal v`: where `v` is a `Value` as explained below.

`Identifier q`: where `q` is a qualified name, given by a non-empty list of strings.

In the remaining cases, the components of the expressions are functional blocks or lists of functional blocks, unless explicitly stated otherwise.

`AndList fbs`: where the elements of `fbs` evaluate to Booleans; the expression denotes the conjunction of these Boolean values.

`OrList fbs`: where the elements of `fbs` evaluate to Booleans; the expression denotes the disjunction of these Boolean values.

`IfThenElse c t f`: the first component, `c`, denotes a Boolean, and `t` and `f` denote objects whose types are the same. The result denotes the value of `t` or `f`, according to the result returned by `c`.

`FunctionCall f as`: usually, `f` denotes a function and `as` a list of function arguments; in this case the expression denotes the function applied to the arguments.

The same notation is used for record or union field access; in this case `f` denotes a record or union object, and `as` is a singleton list containing a record/union field name as an `Identifier` enclosed in a `SimpleFB`.

`RecordCtor fbs`: the list `fbs` denotes a list of record field values.

`UnionCtor e fb`: the expression `e` denotes a field (variant) name, and `fb` denotes the value of that field.

`UnionCase fb e`: this denotes the Boolean result of testing whether the union object `fb` is of the variant given by the expression `e`.

`RestrictType fb t`: as Aldor-- evaluation is multiple-valued, in general (see Section 4) this expression is used to restrict the set of results of `fb` to those belonging to a particular type dentoted by `t`.

`ConvertTo fb t`: is a controlled coercion of `fb` to the type denoted by `t`. In contrast to `RestrictType`, which drops non-conforming values from the result set but preseves the type of those that remain, `ConvertTo` drops non-convertible results of `fb` and changes the type of the rest.

`Add1 fb`: builds an `ADT` from the set of bindings contained in `fb`.

`Add2 fb1 fb2`: builds an `ADT` by extending the ADT built from `fb1` by adding the bindings contained in `fb2`.

`With1 fb`: builds a `CategoryT` from the set of bindings contained in `fb`.

`With2 fb1 fb2`: builds a `CategoryT` by extending the category built from `fb1` by adding the bindings contained in `fb2`.

`VoidExpr`: denotes a void expression that evalutes to `TrivVal`. It is used for technical purposes.

The constructors

> `Literal`, `Identifier`, `FunctionCall`, `IfThenElse`, `RestrictType`, `ConvertTo`, `Add1`, `Add2`, `With1`, `With2`

are *type-forming*. This means that they can be used to construct expressions which can appear in a context where a type is required. For example, they can be used as type expressions in `DTE`s and in the return type of a `FunctionT` type. The constructors

> `RecordCtor`, `UnionCtor`, `RestrictType`, `ConvertTo`

are *callable*. That is, they can be used to form expressions that can appear in the first argument of a `FunctionCall` expression.

3.4 Values

The type `Value` describes the syntax of manifest values in Aldor--. Manifest values are lifted to the expression type, `Expr`, by the constructor `Literal`. Manifest types are included in `Value` by the constructor `TypeVal`. Booleans, integers, floats and strings are given by the corresponding constructors, `BooleanVal` etc. We now explain the other constructors in more detail.

`LambdaVal t fb:` denotes a user-defined function object. The `Type` component `t` will be a `FunctionT` type, as described in the following section. The functional block `fb` denotes the body of the function.

`BuiltInOp str t fVal:` denotes a built-in function, whose name is `str`, whose type is `t` and whose value is given by the Haskell function `fVal`.

`RecordVal vs:` is a manifest record, built from the list of values `vs`.

`UnionVal i v:` is a manifest union value, of variant given by the `Identifier` `i` and with content `v`.

The constructors `LambdaVal`, `BuiltInOp`, `RecordVal` and `UnionVal` are *callable*; that is they can be used to construct the first argument of a `FunctionCall` expression.

`ADT_Val t v:` this is a value of the abstract data type `t`, where `v` gives the representation of the value in the carrier type.

`TrivVal:` is the unique value of type `TrivT`, and also the result of evaluating `VoidExpr`.

3.5 Types

The Haskell data type `Type` describes the syntax of Aldor-- manifest types, which are embedded in the manifest values by the `TypeVal` constructor, and in turn are embedded in `Expr` by the `Literal` constructor. `Type`s also occur directly as components of `ADT_Val`, `LambdaVal` and `BuiltInOp` values.

`TrivT` is the trivial type, with the unique member `TrivVal`. It is also used to represent the proposition '*true*' under the Curry-Howard isomorphism. The `ExitT` type is empty, and is identified with the proposition '*false*'. `TypeT` is the type of all Aldor-- types; Booleans, integers, floats and strings are given by the types `BooleanT` and so forth. We describe the other `Type` constructors in more detail.

`FunctionT ps r:` describes a (dependent) function type with parameters given by `ps` and return type by `r`. It is assumed that the elements of `ps` are `FParmPB`s, each representing a single element of the parameter list; `r` is a `SimpleFB`, encapsulating a type-forming expression.

`BuiltInT ps r`: describes the type of a built-in function; `ps` is the list of argument `Types` and `r` is the result `Type`. Note that the types of built-in functions are *not* dependent, and are given by manifest types rather than functional blocks (as is the case for `FunctionT`).

`RecordT ps`: describes a (dependent) record type, with fields and their types given by `ps`, with the same constraint as the parameters of `FunctionT`.

`UnionT ps`: this is a union type, which is non-dependent in Aldor--. The parameter list `ps` shares the same constraint as `RecordT`.

`ADT fb defTable`: constructs an abstract data type. The first argument, which must be type-valued, provides the carrier type or representation of the ADT. The `defTable` is a `DefTable` that provides a collection of operation signatures together with their implementation over the carrier type.

`CategoryT defTable`: constructs the category defined by the signature contained in the `defTable`.

4 Reduction: evaluation and normalisation

As we said in the introduction, we use the general term *reduction* for the process of computation over Aldor-- terms. Reduction which results in an Aldor-- `Value` is called *evaluation*; otherwise it is known as *normalisation*.

Reduction and type-checking in Aldor-- are closely related. Type expressions in definition table entries (DTEs) need, in general, to be reduced by β- and δ-reduction before a definition body can be type-checked against such an expression.

On the other hand, reduction may require some limited form of type-checking, not to resolve overloading, but to resolve an ambiguity between function application and record/union field access in the syntax of Aldor, which is inherited by Aldor-- since we reuse the Aldor parser in our system. In the concrete syntax, function application is denoted thus `fun(args)`, whereas field access is given by `record.field`; both of these result in the same node in the abstract syntax tree: `FunctionCall`.

As we first noted in Section **??**, it would be impossible to resolve Aldor-- overloading during the process of reduction simply by type-checking components of redexes, since in many cases this would give rise to a non-terminating sequence of reduction and type-checking operations. For instance, consider the example

```
T: Type    == if I > 0                              (1)
              then Boolean
              else Integer;
I: Integer == 1;                                    (2)
I: T       == true;                                 (3)
```

In order to type-check definition `(3)` it is necessary to evaluate `T`, defined by `(1)`. However, the body of `(1)` uses the identifier `I` which is defined by both `(2)` and `(3)`. If the overloading for `I` is to be resolved by type-checking then the definitions of `I` given in both `(2)` and `(3)` will have to be type-checked, thus forming a nonterminating cycle of operations. On the other hand, the example is perfectly legitimate Aldor-- code, and is dealt with using set-valued reduction, as outlined in the remainder of this section.

4.1 The reduction and type-checking environment

The reduction and type-checking environment (RTCE) consists of two parts: a collection of name bindings together with the reduction stack. In Haskell:

```
type RTCE = ( NameBinds , RedStack )
```

The type `NameBinds` was defined in Section **??**, and `RedStack` is a sequence of reduction stack frames. A stack frame associates function arguments with formal function parameters. Function parameters are `FParmFBs` as in the `FunctionT` constructor introduced in Section 3.5. Function arguments will, in general, be the results of previous reductions and are therefore represented by the `RedRes` data type.

```
type RedStack      = [ RedStackFrame ]
type RedStackFrame = FiniteMap FB RedRes
```

Reduction results can take two forms:

```
data RedRes = FullRR NameBinds Value |
              PartialRR NameBinds FB
```

A `FullRR` is the result of an *evaluation* to a value, whereas a `PartialRR` contains a functional block which is the result of *normalisation*. Recall that we also use the term *partial evaluation* for normalisation, and *full evaluation* for evaluation proper.

Both sorts of reduction results also contain a `NameBinds` object; this reflects the fact that in the presence of overloading the name bindings can be affected by the reduction process. The reduction rules in Section 4.2 explain the details of this.

4.2 Reduction rules

The reduction operation is Aldor-- is *set-valued*, to accommodate multiple possible reduction results of of objects containing overloaded identifiers. Thus, this operation in general produces a list of reduction results, `[RedRes]`. The exceptions to this are `DefTables` and `DTEs`: for `DefTable` the result is of type `[(NameBinds, DefTable)]` and for `DTE` it is `[(NameBinds, DTE)]`; note that in these cases, the reduction results are multi-valued as well. More details of this are given in 4.2.2.

In general, reduction takes place in a context, of type `RTCE` as described above. The general form of a reduction statement will be

$$\textit{context} \vdash \textit{before} \rightsquigarrow \textit{after}$$

In every case a modified version of the name bindings component of the context is returned as part of the reduction result.

The reduction algorithm for an Aldor-- object `t` of the form `Ctor` $c_1 \ \ldots \ c_n$ in the context Γ operates as follows:

- Each of the structural components c_i of `t` is reduced in the original context Γ, to yield a list of results rs_i.
- A cross-product xs of all the lists rs_1 to rs_n is constructed. The elements of xs are n-tuples of individual reduction results for the components c_1 to c_n.

  In particular, if some rs_i is an empty list, then the product xs itself will be empty.
- For each tuple in xs, $(x_1, \ldots, x_n)$ say, the name bindings b_i of each of its elements x_i are extracted and are intersected as described in Section **??**.

- If the intersection of the b_is, b say, is not well-formed (i.e. it fails to satisfy the predicate *wellBinds*) then the tuple is not used further in the reduction.
- Otherwise, the final stage of top-level evaluation of the expression `Ctor` $c_1 \ \ldots \ c_n$ can be performed on the tuple $(x_1, \ldots, x_n)$. The intended meaning of the constructor is applied to the result components of x_1 to x_n. For instance, in the case of `AndList`, the intended meaning is the conjunction of a Boolean list.
- The result of this operation can be
  - a failure: in our running example, if one of the x_i is fully-evaluated to something other than a Boolean;
  - a fully-evaluated result: which will happen if (and normally only if) all the x_1 to x_n are fully evaluated; however, in the case of `AndList` the result `False` can be obtained if at least one of the x_i is fully evaluated to `False`;
  - a partially-evaluated result ensues otherwise. This will be built by applying the `Ctor` contstructor to the normalised arguments x_1 to x_n.
- If the binding b is well-formed and the operation result R is not a failure, then the appropriate `RedRes` object – either `FullRR` or `PartialRR` – is constructed from b and R.

- The overall reduction result is the list of `RedRes` objects constructed as above for all tuples in the set xs.

In the following sections we give reduction rules for each of the syntactic meta-types of Aldor--. These rules collectively define the reduction relation denoted by

$$\rightsquigarrow \ .$$

The reduction relation is overloaded to work over all the Aldor-- syntactic meta-types: `FB`, `DefTable`, `DTE`, `Expr`, `Value` and `Type` as well as the semantic meta-type `BindTarg` of identifier binding targets, first introduced in Section **??**.

However, for the simplicity of presentation, we define the reduction relation $\rightsquigarrow$ to be *single-valued*. For a given Aldor-- object, its complete multi-valued reduction result in a given context is therefore the set of all derivations due to $\rightsquigarrow$.

4.2.1 Functional blocks (FBs)

$$\frac{\Gamma \vdash e \rightsquigarrow r}{\Gamma \vdash \texttt{SimpleFB}\, e \rightsquigarrow r}(FB_1) \qquad \frac{\Gamma \vdash fb \rightsquigarrow r}{\Gamma \vdash \texttt{NestedFB}\ \_\ fb \rightsquigarrow r}(FB_2)$$

$$\frac{\Gamma \vdash dte \rightsquigarrow (b, dte')}{\Gamma \vdash \texttt{FParmFB}\, dte \rightsquigarrow (\texttt{PartialRR}\, b\, (\texttt{FParmFB}\, dte'))}(FB_3)$$

$$\frac{}{\Gamma @ (b, \_) \vdash \texttt{VoidFB} \rightsquigarrow [\texttt{PartialRR}\, b\, \texttt{VoidFB}]}(FB_4)$$

Note that in rule (FB_2) the original definition table in the `NestedFB` is not explicitly propagated into the result; rather it was used through the context Γ in the reduction of fb to rs.[5]

The Haskell-style notation $\Gamma @ (b, \_)$ is used for pattern matching components of a context Γ.

[5]Another possible reduction rule for a `NestedFB` is always to normalise it to another `NestedFB`, and thus preserve its definition table. However, this preservation proves to be unnecessary in our implementation of the language.

4.2.2 DTEs and definition tables

Definition table entries:

$$\frac{\Gamma \vdash e \rightsquigarrow er \qquad \Gamma \vdash fb \rightsquigarrow fr \qquad \texttt{wellBinds}\, b}{\Gamma \vdash \texttt{DTE}\, \textit{name}\, fb\, e \rightsquigarrow (b, \texttt{DTE}\, \textit{name}\, fb'\, e')}(DTE)$$

where

$$\begin{array}{lcl} e' & = & \texttt{getExpr}\, er \\ fb' & = & \texttt{getExpr}\, fr \\ b & = & \texttt{getBinds}\, er \otimes \texttt{getBinds}\, fr \end{array}$$

Here the functions `getExpr` and `getBinds` extract the expression and the name bindings from an evaluation result:

```
getExpr (FullRR    _ val)   = SimpleFB (Literal val)
getExpr (PartialRR _ fb )   = getExpr  fb
getExpr (SimpleFB  e)       = e
getExpr (NestedFB  innerFB) = getExpr innerFB
getExpr VoidFB              = error
getExpr FParmFB _           = error

getBinds (FullRR binds _)   = binds
getBinds (PartialRR binds _)= binds
```

Note that in the above definition, the function `getExpr` is overloaded to work on both reduction results and functional blocks.

In the following, we will also use the counter-part of `getBinds` which is called `setBinds`. It is an overloaded function which modifies the name bindings of any object which contains them, e.g. on `RTCE`:

$$\texttt{setBinds} :: \texttt{NameBinds}\, \alpha \rightarrow \alpha$$

Definition tables:

$$\frac{\begin{array}{lcl} \Gamma \vdash d_1 \rightsquigarrow (b_1, d_1') \\ \quad \ldots \\ \Gamma \vdash d_n \rightsquigarrow (b_n, d_n') \end{array} \qquad \begin{array}{l} b = \bigotimes [b_1, \ldots, b_n] \\ \texttt{wellBinds}\, b \end{array}}{\Gamma \vdash \texttt{DefTable}\, [d_1, \ldots, d_n]\, \textit{imports} \rightsquigarrow (b, \texttt{DefTable}\, [d_1', \ldots, d_n']\, \textit{imports})}(DT)$$

4.2.3 Expressions

Literals:
Although an Aldor-- Literal just encapsulates a `Value` object, that object can be complex (e.g. `TypeVal (ADT ...)`), so it may be further reducible:

$$\frac{\Gamma \vdash v \rightsquigarrow r}{\Gamma \vdash \texttt{Literal}\, v \rightsquigarrow r}(E_{Lit})$$

Identifiers:
An identifier is reduced to the reduction results of its binding targets. Since our reduction relation $\rightsquigarrow$

is single-valued, we give here a reduction rule which uses a particular target t. Before the indentifier in question is reduced, the name bindings for it in the environment Γ are reduced to t:

$$\frac{\begin{array}{c}\Gamma@(b,s)\ \vdash\ t\ \in\ \texttt{lookupBinds}\ (b, i@(\texttt{Identifier}\ \_)) \\[1em] \Gamma'@(b\{i \mapsto [t]\}, s)\ \vdash\ t\ \rightsquigarrow\ r\end{array}}{\Gamma\ \vdash\ i\ \rightsquigarrow\ r}(E_{Ident})$$

Conjunction and disjunction:
First of all, note that these logical operations are treated as expressions in Aldor--, not as built-in functions, because they can take a variable number of arguments. They also exhibit a kind of lazy semantics: e.g. the result of `AndList` is `False` if at least one of the operands evaluates to `False`, even if other operands are only partially-evaluated. However, if an operand of a logical expression is fully evaluated, it must evaluate to a Boolean value, otherwise the whole expression fails to reduce:

$$\frac{\begin{array}{lcl}\Gamma & \vdash & fb_1\ \rightsquigarrow\ r_1 \\ & \ldots & \\ \Gamma & \vdash & fb_n\ \rightsquigarrow\ r_n \\[1em] b & = & \bigotimes(\texttt{map getBinds}\ [r_1,\ldots,r_n]) \\ & & \texttt{wellBinds}\ b\end{array} \qquad \begin{array}{l}\texttt{length}\ fulls\ =\ \texttt{length}\ bools \\ fulls\ =\ [f \mid f@(\texttt{FullRR}\ \_\_)\ <-\ [r_1,\ldots,r_n]] \\ bools\ =\ [b \mid \texttt{FullRR}\ \_\ (\texttt{BoolVal}\ b)\ <-\ [r_1,\ldots,r_n]]\end{array}}{\Gamma\ \vdash\ \texttt{AndList}\ [fb_1,\ldots,fb_n]\ \rightsquigarrow\ \texttt{evalAnd}\ [r_1,\ldots,r_n]}(E_{Conj})$$

where the function `evalAnd` is given by

```
evalAnd xs
  | not conj  = [ FullRR    b (BooleanVal False) ]
  | parts==[] = [ FullRR    b (BooleanVal True ) ]
  | otherwise = [ PartialRR b (SimpleFB (AndList (map getFB parts))) ]
  where
  parts = [part | part@(PartialRR _ _) <- [r1,...,rn]]
  conj  = and bools
```

The reduction rule for `OrList` has the dual definition.

Conditionals:
If the condition *cond* is fully evaluated to a boolean value, then the corresponding branch of the `IfThenElse` expression is reduced. If the condition is only partially evaluated, then the branches are not reduced at all:

$$\frac{\Gamma\ \vdash\ cond\ \rightsquigarrow\ \texttt{FullRR}\ b\ (\texttt{BooleanVal True}) \qquad (\texttt{setBinds}\ b\ \Gamma)\ \vdash\ then\ \rightsquigarrow\ tr}{\Gamma\ \vdash\ (\texttt{IfThenElse}\ cond\ then\ \_)\ \rightsquigarrow\ tr}(E_{Cond1})$$

$$\frac{\Gamma\ \vdash\ cond\ \rightsquigarrow\ \texttt{FullRR}\ b\ (\texttt{BooleanVal False}) \qquad (\texttt{setBinds}\ b\ \Gamma)\ \vdash\ else\ \rightsquigarrow\ er}{\Gamma\ \vdash\ (\texttt{IfThenElse}\ cond\ \_\ else)\ \rightsquigarrow\ tr}(E_{Cond1})$$

$$\frac{\Gamma\ \vdash\ cond\ \rightsquigarrow\ \texttt{PartialRR}\ b\ cfb}{\Gamma\ \vdash\ (\texttt{IfThenElse}\ cond\ then\ else)\ \rightsquigarrow\ \texttt{PartialRR}\ b\ (\texttt{SimpleFB}\ (\texttt{IfThenElse}\ cfb\ then\ else))}(E_{Cond3})$$

Function call:
The *callable* is reduced first. The result can be:

1. a (fully-evaluated) user-defined lambda object (`LambdaVal`);
2. a (fully-evaluated) built-in function (`BuiltInOp`);
3. a (fully-evaluated) record object (`RecordVal`);
4. a (fully-evaluated) union object (`UnionVal`);
5. a (partially-evaluated) record constructor expression (`RecordCtor`);
6. a (partially-evaluated) union constructor expression (`UnionT`);
7. any other partially-evaluated result.

Note that in Aldor--, function applications (of user-defined lambda objects and of built-in functions) have the same abstract syntax, namely `FunctionCall`, as record or union access operations. This makes reduction of `FunctionCall` expressions quite complex, especially in the last case listed above (7).

Consider application of user-defined lambda objects first. The actual arguments are all reduced and pushed on the stack, after that the function body is reduced in the new environment:

$$\frac{\begin{array}{lll} \Gamma & \vdash & \mathit{callable} \leadsto \texttt{FullRR}\, b_0\, (\texttt{LambdaVal}\, (\texttt{FunctionT}\, [p_1, \ldots, p_n]\ \_)\ \mathit{body}) \\ \Gamma & \vdash & a_1 \leadsto r_1 \\ & \ldots & \\ \Gamma & \vdash & a_n \leadsto r_n \\ \\ b & = & \bigotimes [b_0,\ \texttt{getBinds}\, r_1,\ \ldots,\ \texttt{getBinds}\, r_n]\ , \qquad \texttt{wellBinds}\, b \\ \Gamma' & = & \texttt{pushArgs}\, [r_1\ \ldots\ r_n]\ [p_1\ \ldots\ p_n]\ \Gamma \\ \\ \Gamma' & \vdash & \mathit{body} \leadsto \mathit{br} \end{array}}{\Gamma \vdash (\texttt{FunctionCall}\, \mathit{callable}\ [a_1, \ldots, a_n]) \leadsto \mathit{br}} (FCall_1)$$

Here the `pushArgs` function associates, on the reduction stack of Γ, the formal parameters $p_1, \ldots, p_n$ with the reduction results $r_1, \ldots, r_n$ of the actual arguments $a_1, \ldots, a_n$. Note that the arity of the formal parameters and the actual arguments must be the same:

$$\texttt{pushArgs}\, [r_1\ \ldots\ r_n]\ [p_1\ \ldots\ p_n]\ \Gamma@(b, s) = (b,\ s \cup \{p_1 \mapsto r_1, \ldots, p_n \mapsto r_n\})\ .$$

The reduction rule for a built-in function application is somewhat similar, but the difference is that a run-time type-check of the actual arguments is performed in this case. The check only applies to those arguments which are fully evaluated. If at least one of the arguments is not fully evalutaed, a built-in function application reduces to a normal form. Also note that since bult-in functions in Aldor-- are not recursive, they can be applied directly, without modifying the reduction stack:

$$\frac{\begin{array}{lcl} \Gamma & \vdash & \mathit{callable} \ \rightsquigarrow \ \texttt{FullRR}\, b_0\, BI@(\texttt{BuiltInOp}\, \mathit{name}\, (\texttt{BuiltInT}\, [t_1, \ldots, t_n]\, \_)\, \mathit{body}) \\ \Gamma & \vdash & a_1 \ \rightsquigarrow \ r_1 \\ & \ldots & \\ \Gamma & \vdash & a_n \ \rightsquigarrow \ r_n \\ \\ b & = & \bigotimes[b_0,\ \texttt{getBinds}\, r_1,\ \ldots,\ \texttt{getBinds}\, r_n]\ , \quad \texttt{wellBinds}\, b \\ \\ & & \texttt{conformsTo}\, r_1\, t_1 \\ & \ldots & \\ & & \texttt{conformsTo}\, r_n\, t_n \\ \\ br & = & \texttt{applyBI}\, \mathit{body}\, [r_1, \ldots, r_n]\, b \end{array}}{\Gamma \vdash (\texttt{FunctionCall}\, \mathit{callable}\, [a_1, \ldots, a_n]) \ \rightsquigarrow \ br} (FCall_2)$$

Here the `conformsTo` function implements a "liberal" run-time type check. The check is facilitated by the fact that the parameters of built-in functions always have simple manifest types, which can only be `BooleanT`, `IntegerT`, `FloatT` or `StringT`. If the type of the left-hand side (the reduction result) cannot easily be inferred, e.g. if the reduction result is a normal form, `conformsTo` assumes that the type conformance is satisfied. This reduction strategy of built-in functions can produce extraneous results, but the idea behind the reduction and type-checking algorithm of Aldor-- is that such results will at some later point be eliminated, otherwise an ambiguity will be detected. On the other hand, the stragegy of "liberal" run-time type checks guarantees that no result will be lost unnecessarily:

```
conformsTo (FullRR _ (BooleanVal _)) BooleanT = True
conformsTo (FullRR _ (IntegerVal _)) IntegerT = True
conformsTo (FullRR _ (FloatVal   _)) FloatT   = True
conformsTo (FullRR _ (StringVal  _)) StringT  = True
conformsTo (FullRR _ _)              _        = False
conformsTo (PartialRR _ _)           _        = True
```

The `applyBI` function used above actually applies the *body* of a built-in function to the arguments $[r_1, \ldots, r_n]$, if all of them are fully-evaluated. Otherwise, it produces a partially-evaluated result. Thus,

$$\texttt{applyBI}\, \mathit{body}\, [\texttt{FullRR}\, \_v_1,\ \ldots, \texttt{FullRR}\, \_v_n]\, b \ = \ \texttt{FullRR}\, b\, (\mathit{body}\ v_1\ \ldots\ v_n)$$

$$\texttt{applyBI}\, \mathit{body}\, [r_1, \ldots, r_n]\, b \ = \ \texttt{PartialRR}\, b\, (\texttt{SimpleFB}\, (\texttt{FuncionCall}\, BI\, [\texttt{rrToFB}\, r_1,\ \ldots,\ \texttt{rrToFB}\, r_n]))$$

where BI is the same as in the premises of rule ($FCall_2$), and the function `rrToFB` converts a `RedRes` object into a functional block:

```
rrToFB (FullRR    _ val) = SimpleFB (Literal val)
rrToFB (PartialRR _  fb) = fb
```

Let us now consider the cases when `FunctionCall` actually denotes record field access.

$$\frac{\begin{array}{l} [(rt_1, i_1), \ldots, (rt_m, i_m)] \ = \ \texttt{findRecordTypes}\, \Gamma\, \mathit{ident} \\ res_1 \ = \ \texttt{getRecordField}\, \Gamma\, rt_1\, rv\ \ i_1 \\ \ldots \\ res_m \ = \ \texttt{getRecordField}\, \Gamma\, rt_m\, rv\ \ i_m \end{array}}{(\texttt{FullRR}\, \_\ rv@(\texttt{RecordVal}\, \_),\ \mathit{ident}@(\texttt{Identifier}\, [\_])) \ \blacktriangleright_{FC} \ \texttt{concat}\, [res_1, \ldots, res_m]} (\blacktriangleright_{FC\ 3})$$

Definitions of `findRecordTypes` and `getRecordField.` are standard.

Function call: other cases

Reduction is defined similarly.

Other cases of expressions

Reduction here is standard.

4.2.4 Values

Reduction here is standard.

4.2.5 Types

Reduction here is standard.

5 The type system of Aldor--

5.1 The principles

In Aldor-- types are associated with

- types;
- values;
- expressions;
- definition table entries (DTEs);
- identifier binding targets; and
- functional blocks (FBs).

Note that types are *not* associated with definition tables.

In general, all Aldor-- objects which are defined at the definition-table level have explicit type information contained in their definition table entries (DTEs). The type-checking algorithm is, in most cases, based on retrieving this type information from the DTEs.

In Aldor--, definition table entries are objects of the Haskell type

```
data  DTE = DTE  String body typeExpr
```

By type-checking of a DTE we mean the process of verifying that the `body` has the type denoted by the `typeExpr`. The `body` here is also known as the *left-hand side (LHS)*, and the `typeExpr` is known as the *right-hand side (RHS)* or the *type-checking target*. We generalise the terminology of left- and right-hand sides to the type-checking of other Aldor-- constructs, such as checking function arguments (the left-hand side) against formal parameters (the right).

The type-checking algorithm for Aldor-- operates in three modes.

N mode. In this case the right-hand side is absent. There is thus no target to type-check against, and so the algorithm checks the left-hand side for the internal typing integrity. For some kinds of left-hand sides (such as stand-alone Aldor-- expressions encapsulated in `SimpleFBs`, or definition tables), the N mode is the only possible one.

Usually, but not always, type-checking in the N mode also produces the *inferred type* of the LHS. Apart from the definition tables, the cases when the inferred type is not generated include expressions constructed by `IfThenElse`, `RecordCtor` and `UnionCtor`, and values constructed by `RecordVal` and `UnionVal`, as such objects do not contain enough information for accurate type inference.

T mode. In this mode the right-hand side is a manifest type. In most cases type-checking here also proceeds by inferring the type of the LHS (via the N mode) and then by checking the equivalence of the inferred and the target type. However, in cases (pointed out above) when the type of the LHS cannot be accurately inferred, specific T mode algorithms are used instead.

E mode. In this final case, the RHS is a *normalised* type-forming expression, constructed by `Identifier`, `FunctionCall` and `IfThenElse`. More details are given in Section **??**.

The environment for type-checking is given by the Haskell data type `RTCE` introduced in Section 4.1. This is the same environment that is used in reduction.

As was said earlier, Aldor-- permits the overloading of identifiers. During the process of type-checking an expression `e` containing an overloaded identifier, `x` say, some bindings of `x` can be found to be inconsistent with type required for `x` in the context of `e`. Because of this, the type-checking operation returns modified name bindings from which all inconsistent bindings have been removed.

The type-checking operation may also return the inferred type of the LHS. However, this is not a necessary condition for successful type-checking in every case: exceptions include the special cases of type-checking in the N mode mentioned above.

In the presence of overloading, type-checking can return multiple results, and so the result of type-checking is given by a list of `TCRes`:

```
type TCRes = ( NameBinds , Maybe Expr )
```

A failure of type-checking is signalled by an empty list of results.

The type of each definition table entry should be unique, since otherwise the definition itself is ambiguous. So, in the case of DTEs, type-checking is also said to fail when a non-singleton list is produced. This uniqueness requirement makes the type system *non-monotonic*: if a new definition of an existing identifier is introduced, then it is possible that an existing DTE would become ambiguous, and thus fail to type-check.

In the remainder of this section we present, in turn, the type rules for functional blocks, identifier binding targets, DTEs, expresssions, values and types themselves. The type rules collectively define the relation

$$\Gamma \vdash x : t;\, b \tag{1}$$

which means that *"the object x has a type given by the type-valued object t under the name bindings b in the context Γ"*, where Γ is of the Haskell type `RTCE`. The name bindings b in the right-hand side of the typing relation is always a *restriction* of the name bindings contained in Γ.

In other words, relation 1 holds if any only if the result of type-checking of x against t, say `tcrs`, which is of Haskell type `[TCRes]`, contains a pair `(`b`, `t'`)`. Since there is an explicit target type t, type-checking of x is done in the T or E mode; t', is the inferred type for x. Note that the inferred type t' is not always the same as the target type t. In general, due to the sub-typing relation between some of the Aldor-- types (see Section 5.7), it is not even true that the normalisations of t and t' must be equivalent. However, the inferred type t' has the property that x must type-check against it under the same name bindings:

$$\Gamma \vdash x : t';\ b\,.$$

The typing relation ":" introduced by 1 is single-valued; it associates a single type t with the object x. In general, as stated above, x can be characterised by multiple types. The set $\mathcal{T}(x)$ of all types (endowed with the corresponding name bindings) for x is given by all derivations from the Aldor-- type rules:

$$\mathcal{T}(x) \;=\; \{(b,t) \mid \Gamma \vdash x : t;\ b\}\,,$$

and the set of all types which can be *inferred* for x in all possible type-checking operations, is a sub-set of $\mathcal{T}(x)$.

The type-checking algorithm can use the typing relation 1 for type-checking of the left-hand side in the T or E mode, or for inferring the type of the left-hand side in the N mode. However, if the type of the left-hand side cannot be inferred in the N mode (either because there is insufficient information for type inference, or the LHS is a definition table), another typing relation is required:

$$\texttt{wellTyped}\,x;\ b \tag{2}$$

which means that x type-checks successfully for internal integrity under the restricted name bindings b.

The type rules presented in Sections 5.2–5.7 define both typing relations 1 and 2.

5.2 Functional blocks (FBs)

The type rules for Aldor-- functional blocks are given below:

$$\frac{\Gamma \vdash e : t;\ b}{\Gamma\ (\texttt{SimpleFB}\,e) : t;\ b}(SFB:) \qquad \frac{\begin{array}{lll}\Gamma & \vdash & fb : t;\ b_1 \qquad\qquad b = b_1 \otimes b_2\\ \Gamma & \vdash & \texttt{wellTyped}\,dt;\ b_2 \qquad \texttt{wellBinds}\,b\end{array}}{\Gamma \vdash (\texttt{NestedFB}\,dt\ fb) : t;\ b}(NFB:)$$

$$\frac{\Gamma \vdash dte : t;\ b}{\Gamma \vdash (\texttt{FParmFB}dte) : t;\ b}(FPFB:) \qquad \frac{}{\Gamma \vdash \texttt{VoidFB} : \texttt{TrivT};\ \texttt{getBinds}\,\Gamma}(VFB:)$$

5.3 DTEs and definition tables

In a DTE, the body is type-checked against a *reduced* type expression, which must be indeed type-valued, in the context containing the name bindings returned by that particular reduction. If the check suceeds, we can associate both the original and the reduced type with the definition body:

$$\frac{\Gamma \vdash te : Type;\ b_1 \qquad \Gamma_1 \vdash te \rightsquigarrow nte;\ b_2 \qquad \Gamma_2 \vdash fb : nte;\ b_3}{\Gamma \vdash (\texttt{DTE}\,te\ fb) : te;\ b_3}(DTE:)$$

where $\Gamma_1 = \texttt{setBinds}\,b_1\,\Gamma$ and $\Gamma_2 = \texttt{setBinds}\,b_2\,\Gamma_1$,

A definition table is well-typed if all its DTEs type-check under compatible name bindings:

$$\frac{\begin{array}{lll}\Gamma & \vdash & dte_1 : t_1;\ b_1 \qquad\qquad b = \bigotimes[b_1,\ \ldots,\ b_m]\\ & \ldots & \\ \Gamma & \vdash & dte_m : t_n;\ b_m \qquad\quad \texttt{wellBinds}\,b\end{array}}{\Gamma \vdash \texttt{wellTyped}(\texttt{DefTable}\{name_1 \mapsto dte_1,\ \ldots,\ name_m \mapsto dte_m\}_{-});\ b}(DT:)$$

5.4 Identifier binding targets

Both `DefTargs` and FPTargs encapsulate definition table entries, and are similar from the type-checking point of view; the difference between them lies in the semantics of name bindings construction (Section **??**) and reduction (Section 4.2.3):

$$\frac{\Gamma \vdash \mathit{dte} \ : \ t;\ b}{\Gamma \vdash (\texttt{DefTarg}\,\mathit{dte}) \ : \ t;\ b}(DTarg :) \qquad \frac{\Gamma \vdash \mathit{dte} \ : \ t;\ b}{\Gamma \vdash (\texttt{FPTarg}\,\mathit{dte}) \ : \ t;\ b}(FPTarg :)$$

5.5 Expressions

5.5.1 Literal

A `Literal` has the same type as the value encapsulated in it:

$$\frac{\Gamma \vdash v \ : \ t;\ b}{\Gamma \vdash (\texttt{Literal}\,v) \ : \ t;\ b}(\mathit{Literal} :)$$

5.5.2 Identifier

The type of an `Identifier` is determined by its particular binding target being considered, under the name bindings restricted to that target:

$$\frac{\begin{array}{lll}\Gamma & \vdash & \mathit{targ} \in \texttt{lookupBinds}(\texttt{getBinds}\,\Gamma)\ (\mathit{ident}@(\texttt{Identifier}\ \_)) \\ \Gamma' & \vdash & \mathit{targ} \ : \ t;\ b\end{array}}{\Gamma \vdash \mathit{ident} \ : \ t;\ b}(\mathit{Ident} :)$$

where $\Gamma' = \texttt{modifyBinds}\ (\mathit{ident} \mapsto [\mathit{targ}])\ \Gamma$.

5.5.3 AndList, OrList

We give the type rule for `AndList` only; the rule for `OrList` is similar. All operands of `AndList` must be Booleans, and the result is a Boolean as well:

$$\frac{\begin{array}{lll}\Gamma & \vdash & fb_1 \ : \ \texttt{BooleanT};\ b_1 \\ & \dots & \\ \Gamma & \vdash & fb_n \ : \ \texttt{BooleanT};\ b_n\end{array} \qquad \begin{array}{l} b = \bigotimes[b_1,\ \dots,\ b_n] \\ \texttt{wellBinds}\ b\end{array}}{\Gamma \vdash (\texttt{AndList}\,[fb_1,\ \dots,\ fb_n]) \ : \ \texttt{BooleanT};\ b}(\mathit{AndList} :)$$

5.5.4 IfThenElse

There are three type rules here. The first one states that the conditional expression is well-typed if the condition if of type Boolean and the branches are of equivalent types. This rule is used by the N mode type-checking of the conditional expression; it does not allow us to infer the type of the whole expression, since we may not be able to unify the types of the branches:

$$\frac{\begin{array}{lll}\Gamma & \vdash & \mathit{cond} \ : \ \texttt{BooleanT};\ b_0 \\ \Gamma & \vdash & \mathit{then} \ : \ t_1;\ b_1 \\ \Gamma & \vdash & \mathit{else} \ : \ t_2;\ b_2\end{array} \qquad \begin{array}{l}\Gamma \vdash t_1 \equiv t_2 \\ b = \bigotimes[b_0, b_1, b_2] \\ \texttt{wellBinds}\ b\end{array}}{\Gamma \vdash \texttt{wellTyped}\ (\texttt{IfThenElse}\ \mathit{cond}\ \mathit{then}\ \mathit{else});\ b}(ITE :_1)$$

The second rule is used by the type-checking algorithm operating in the T mode, that is, when there is a single target type t for both branches. In this case, the inferred type of the conditional expression will also be t:

$$\frac{\begin{array}{lllll} \Gamma & \vdash & cond : \texttt{BooleanT};\ b_0 & & b = \bigotimes[b_0, b_1, b_2] \\ \Gamma & \vdash & then : t;\ b_1 & & \\ \Gamma & \vdash & else : t : b_2 & & \texttt{wellBinds}\, b \end{array}}{\Gamma \vdash (\texttt{IfThenElse}\, cond\, then\, else) : t;\ b}\ (ITE :_2)$$

The third rule is a special case when the target type t is a type-valued conditional expression itself, so the second rule is not applicable. In this case, the conditions of the LHS and the RHS must be equivalent, and both branches of the LHS must type-check against the corresponding branches of the RHS. This rule is used by the type-checking algorithm operating in the E mode:

$$\frac{\begin{array}{llllll} \Gamma & \vdash & rhs@(\texttt{IfThenElse}\, rcond\, rthen\, relse) : Type;\ b_1 & \Gamma & \vdash & lcond \equiv rcond \\ \Gamma & \vdash & lcond : \texttt{BooleanT};\ b_2 & b & = & \bigotimes[b_1, b_2, b_3, b_4] \\ \Gamma & \vdash & lthen : rthen;\ b_3 & & & \\ \Gamma & \vdash & lelse : relse;\ b_4 & & & \texttt{wellBinds}\, b \end{array}}{\Gamma \vdash (\texttt{IfThenElse}\, lcond\, lthen\, lelse) : rhs;\ b}\ (ITE :_3)$$

5.5.5 FunctionCall

As mentioned earlier (Section 4.2.3), in Aldor-- the `FunctionCall` expression can denote application of a user-defined function (lambda object), application of a built-in function, record field access, or union field access, depending on the type of the callable object.

If the callable has type `FunctionT` or `RecordT`, so the `FunctionCall` is a lambda object application or a record field access, then the type of the callable can be dependent. For a `FunctionT` type

$$\texttt{FunctionT}\ [p_1, \ldots, p_N]\ rt\ ,$$

where $p_1, \ldots, p_N$ are `FParmFB`s denoting the formal function parameters, and rt is a `SimpleFB` denoting the function return type, the type expression of p_i ($i = 2, \ldots, N$) can contain *dependent identifiers*, that is, identifiers bound to the names declared in $p_1, \ldots, p_{i-1}$.

In other words, the type of the parameter p_i can depend on the *values* of the actual arguments $a_1, \ldots, a_{i-1}$ corresponding to the parameters $p_1, \ldots, p_{i-1}$. The return type rt can also contain dependent identifiers (bound to any of the $p_1, \ldots, p_N$), and it can thus depend of any of the arguments $a_1, \ldots, a_N$.

Conversely, if the parameter p_j ($j = 1, \ldots, N$) declares a name to which any of the identifiers occurring in $p_{j+1}, \ldots, p_N, rt$ are bound, we will call the parameter p_j to be *depended-upon*.

Therefore, in order to type-check a `FunctionCall` expression with the callable being of a (potentially dependent) `FunctionT` type, we first need to evaluate the types of the parameters $p_1, \ldots, p_N$ in the dependent environment. The dependent environment Γ' is formed from the original reduction and type-checking environment Γ by reducing the actual call arguments (or actual record fields) a_j corresponding to the depended-upon function parameters p_j, and pushing the reduction results on the stack. This is done by the Haskell function

$$\texttt{smartPush}\ [a_1, \ldots, a_N]\ [p_1, \ldots\ p_N, rt]\ \Gamma\ .$$

Generally speaking, since reductions are multi-valued in Aldor--, the `smartPush` function produces a set of new environments $\{\Gamma'\}$ for a single original environment Γ. This set can be defined via the following derivation rule for each individual Γ':

$$\frac{\begin{array}{ll} \begin{array}{l} \Gamma \ = \ (b, s) \\ \texttt{isDependedUpon}\, p_{j_1}\ [p_{j_1+1}, \ldots, p_N, rt]\ \Gamma \\ \ldots \\ \texttt{isDependedUpon}\, p_{j_K}\ [p_{j_K+1}, \ldots, p_N, rt]\ \Gamma \end{array} & \begin{array}{l} \Gamma \vdash a_{j_1} \ \rightsquigarrow \ r_{j_1} \\ \ldots \\ \Gamma \vdash a_{j_K} \ \rightsquigarrow \ r_{j_K} \\ b \ = \ \bigotimes[\texttt{getBinds}\, r_{j_1},\ \ldots,\ \texttt{getBinds}\, r_{j_K}] \\ \texttt{wellBinds}\, b \end{array} \end{array}}{\begin{array}{lll} \Gamma' & = & (b,\ [\{p_{j_1} \mapsto r_{j_1}, \ldots, p_{j_K} \mapsto r_{j_K}\}]\ ++\ s) \\ \Gamma' & \in & \texttt{smartPush}\,[a_1, \ldots, a_N]\ [p_1, \ldots\ p_N, rt]\ \Gamma \end{array}} (smartPush) \tag{3}$$

Here the predicate

$$\texttt{isDependedUpon}\, p\ [q_1, \ldots, q_M]\ \Gamma$$

is true iff there exists index k ($1 \leq k \leq M$) and the identifier i contained in the functional block q_k, such that

$$p \in (\texttt{lookupBinds}\ (\texttt{getBinds}\ \Gamma)\ i)\ .$$

The function given by rule (3) is called `smartPush` because it "intelligently" reduces not all the actual arguments $[a_1, \ldots, \_N]$ (that would be wasteful), but only those which correspond to the depended-upon function parameters. If there are no such parameters, then `smartPush` produces a single environment $\Gamma' = \Gamma$.

We can now present the type rule for the application of a user-defined function. The `FunctionCall` arguments are checked against the formal parameters in the dependent environment Γ', and the inferred type of the application is the reduced function return type rt':

$$\frac{\begin{array}{lll} \Gamma & \vdash & \mathit{callable}\ :\ (\texttt{FunctionT}\,[p_1, \ldots, p_N]\ rt);\ b_0 \\ \Gamma' & = & \texttt{smartPush}\,[a_1, \ldots, a_N]\ [p_1, \ldots, p_N, rt]\ (\texttt{setBinds}\, b_0\, \Gamma) \\ & & \\ \Gamma' & \vdash & [a_1, \ldots, a_N]\ :_{dep}\ [p_1, \ldots, p_N];\ b \\ \Gamma'' & = & \texttt{setBinds}\, b\, \Gamma' \\ \Gamma'' & \vdash & rt \ \rightsquigarrow \ rt' \end{array}}{\Gamma \vdash (\texttt{FunctionCall}\, \mathit{callable}\ [a_1, \ldots, a_N])\ :\ \texttt{getExpr}\, rt';\ \texttt{getBinds}\, rt'} (FC :_1)$$

Note that the function `getExpr` is required in the above rule to convert rt' (an object of the meta-type `RedRes`) into an `Expr`, so it can be used in the context where a type is required.

The relation $:_{dep}$ which was used above means that its LHS (a vector of any `FB`s) has the dependent vector type given by the RHS (a vector of `FParmFB`s). This relation is defined by the following rule:

$$\frac{\begin{array}{ll} \begin{array}{lll} \Gamma & \vdash & (\texttt{getType}\, p_1) \ \rightsquigarrow \ r_1 \\ & \ldots & \\ \Gamma & \vdash & (\texttt{getType}\, p_N) \ \rightsquigarrow \ r_N \\ & & \\ b_1 & = & \texttt{getBinds}\, r_1 \\ & \ldots & \\ b_N & = & \texttt{getBinds}\, r_N \end{array} & \begin{array}{lll} b & = & \bigotimes[b_1, \ldots, b_N],\ \ \texttt{wellBinds}\, b \\ \Gamma' & = & \texttt{setBinds}\, b\, \Gamma \\ & & \\ \Gamma' & \vdash & a_1\ :\ (\texttt{getExpr}\, r_1);\ b'_1 \\ & \ldots & \\ \Gamma' & \vdash & a_N\ :\ (\texttt{getExpr}\, r_N);\ b'_N \\ b' & = & \bigotimes[b'_1, \ldots, b'_N],\ \ \texttt{wellBinds}\, b' \end{array} \end{array}}{\Gamma \vdash [a_1, \ldots, a_N]\ :_{dep}\ [p_1, \ldots, p_N];\ b'} (DepVec)$$

Here the function `getType :: FB -> Expr` returns the type expression of the DTE encapsulated in an `FParmFB`:

```
getType (FParmFB (DTE _ _ typeExpr)) = typeExpr
```

Similarly, in a RecordT type

$$\texttt{RecordT}\,[p_1,\ldots,p_N]$$

the type of any field p_i ($i = 2,\ldots,N$) can contain dependent identifiers bound to $p_1,\ldots,p_{i-1}$. Thus, the type of p_i can depend on the *values* of the fields $a_1,\ldots,a_{i-1}$ in the corresponding *reduced* callable object which would be given by

$$\begin{array}{lll} \Gamma & \vdash & \mathit{callable} \;\leadsto\; \texttt{FullRR}\,b_0\,(\texttt{RecordVal}\,[a_1,\ldots,a_N]) \\ & \text{or} & \\ \Gamma & \vdash & \mathit{callable} \;\leadsto\; \texttt{PartialRR}\,b_0\,(\texttt{RecordCtor}\,[a_1,\ldots,a_N])\,. \end{array} \tag{4}$$

The field p_j ($j = 1,\ldots,N-1$) is called *depended-upon* if any of the subsequent field declarations $p_{j+1},\ldots,p_N$ contains an identifier bound to p_j.

The function `smartPush` for a `RecordT` is defined by a rule similar to 3 above, with the exception that it does not involve the functional block rt. The `FunctionCall` expression (which in this case denotes a record field access) has a single argument which must be an identifier acting as a record field selector. The type of `FunctionCall` is in this case the type of the selected record field reduced in the dependent environment Γ'':

$$\frac{\begin{array}{lll} \Gamma' & = & \texttt{setBinds}\,b_0\,\Gamma \\ \Gamma' & \vdash & \mathit{callable}\;:\;(\texttt{RecordT}\,[p_1,\ldots,p_N]\;rt);\;b_1 \\ \exists & k. & (\texttt{FPTarg}\,p_k) \in (\texttt{lookupBinds}\,b_1\,(i@(\texttt{Identifier}\;\_))) \\ \Gamma'' & = & \texttt{smartPush}\,[a_1,\ldots,a_N]\,[p_1,\ldots,p_N]\,(\texttt{setBinds}\,b_1\,\Gamma) \\ \Gamma'' & \vdash & (\texttt{getType}\,p_k)\;\leadsto\;t' \end{array}}{\Gamma \vdash (\texttt{FunctionCall}\,\mathit{callable}\,[i])\;:\;t';\;\texttt{getBinds}\,t'}\,(FC\,:_2)$$

In the type rule $(FC\;:_2)$ above, the name bindings b_0 and the actual record fields $[a_1,\ldots,a_N]$ are determined by pattern-matching rule 5. If, however, the pattern-matching fails (that is, the $\mathit{callable}$ cannot be reduced to either `RecordVal` or `RecordCtor`, e.g. it is normalised to an `Identifier`), then we take $b_0 = \texttt{getBinds}\;\Gamma$, $\Gamma' = \Gamma'' = \Gamma$. This means that no dependent environment is formed in that case, and the dependent identifiers occurring in the type expressions of $p_1,\ldots,p_N$ would be normalised to themselves.

Note that, unlike the case of lambda object application, type-checking of a dependent record field access expression requires reduction of the callable object.

In the remaining two cases, when the callable object is a **built-in function** or a **union**, the type of the callable cannot be dependent. Furthermore, the parameter types and the return type of a bult-in function are always given by elementary manifest types which do not require any reduction, so the type rule for the built-in function application is straightforward:

$$\frac{\begin{array}{lll} \Gamma & \vdash & \mathit{callable}\;:\;(\texttt{BuiltInT}\,[t_1,\ldots,t_N]\;rt);\;b_0 \\ \Gamma' & = & \texttt{setBinds}\,b_0\,\Gamma \\ \Gamma' & \vdash & a_1\;:\;t_1;\;b_1 \\ & \ldots & \\ \Gamma' & \vdash & a_N\;:\;t_N;\;b_N \\ b & = & \bigotimes[b_1,\ldots,b_N],\quad \texttt{wellBinds}\,b \end{array}}{\Gamma \vdash (\texttt{FunctionCall}\,\mathit{callable}\,[a_1,\ldots,a_N])\;:\;rt;\;b}\,(FC\,:_3)$$

However, if the callable has a `UnionT` type, there is a further restriction that the union object (constructed by a `UnionVal` or a `UnionCtor`) must be tagged with the same identifier as the argument of the `FunctionCall`. Thus, in this case the callable has to be reduced and its tag extracted. If the tag cannot be obtained (because the callable does not reduce to either a `UnionVal` or a `UnionCtor`), type-checking fails:

$$\frac{\begin{array}{lcl} \Gamma & \vdash & \mathit{callable} \ : \ (\texttt{UnionT}\,[p_1,\ldots,p_N]);\ b_1 \\ \exists & k. & (\texttt{FPTarg}\,p_k) \in (\texttt{lookupBinds}\,b_1\ (i@(\texttt{Identifier}\ \_))) \\ \Gamma' & = & \texttt{setBinds}\,b_1\,\Gamma \\ \Gamma' & \vdash & \mathit{callable}\ \leadsto\ rc \\ \Gamma'' & = & \texttt{setBinds}\,(\texttt{getBinds}\,rc)\,\Gamma' \\ \Gamma'' & \vdash & (\texttt{getTag}\,rc)\ \equiv\ i \\ \Gamma'' & \vdash & (\texttt{getType}\,p_k)\ \leadsto\ t \end{array}}{\Gamma \vdash (\texttt{FunctionCall}\,\mathit{callable}\,[i])\ :\ t;\ \texttt{getBinds}\,t}(FC :_4)$$

Here the function `getTag :: RedRes -> Expr` is given by the equations

```
getTag (FullRR    _ (UnionVal tag _)) = tag
getTag (PartialRR _ (SimpleFB (UnionCtor tag _))) = tag
```

5.5.6 RecordCtor

A `RecordCtor` expression can be type-checked in the N mode or the T mode. However, type inference in the N mode may be inaccurate (too general): we would not be able to infer a dependent record type because there is no information about the dependencies (i.e. constraints on the fields' types) in the record constructor:

$$\frac{\begin{array}{lcl} \Gamma & \vdash & a_1\ :\ t_1;\ b_1 \\ & \ldots & \\ \Gamma & \vdash & a_N\ :\ t_N;\ b_N \end{array} \qquad \begin{array}{l} b\ =\ \bigotimes[b_1,\ldots,b_N] \\ \texttt{wellBinds}\,b \end{array}}{(\texttt{RecordCtor}\,[a_1,\ldots,a_N])\ :\ (\texttt{RecordT}\,[t_1,\ldots,t_N]);\ b}(RCtor :_1)$$

In the T mode, if the target record type is dependent, we use the `smartPush` function to construct the new type-checking environment (cf. Section 5.5.5), and type-check the record constructor arguments in that environment against the declared field types:

$$\frac{\begin{array}{lcl} \Gamma' & = & \texttt{smartPush}\,[a_1,\ldots,a_N]\,[p_1,\ldots,p_N]\,\Gamma \\ \Gamma' & \vdash & [a_1,\ldots,a_N]\ :_{dep}\ [p_1,\ldots,p_N];\ b \end{array}}{(\texttt{RecordCtor}\,[a_1,\ldots,a_N])\ :\ (\texttt{RecordT}\,[t_1,\ldots,t_N]);\ b}(RCtor :_2)$$

5.5.7 UnionCtor

In the N mode, we can only type-check the components of a `UnionCtor` expression for internal consistency; the type of the expression cannot be inferred since we have no information about all possible members of the union:

$$\frac{\begin{array}{lcl} \Gamma & \vdash & \texttt{wellTyped}\,i@(\texttt{Identifier}\ \_);\ b_1 \\ \Gamma & \vdash & \texttt{wellTyped}\,f;\ b_2 \end{array} \qquad \begin{array}{l} b\ =\ b_1 \otimes b_2 \\ \texttt{wellBinds}\,b \end{array}}{\Gamma \vdash \texttt{wellTyped}\,(\texttt{UnionCtor}\,i\ f);\ b}(UCtor :_1)$$

In the T mode, when the target union type is known, we check that the selector identifier (first argument of the `UnionCtor`) is indeed bound to a field of that union, and the second argument of `UnionCtor` satisfies that type:

$$\frac{\begin{array}{l}\exists \quad k. \quad (\texttt{FPTarg}\, p_k) \in (\texttt{lookupBinds}\,(\texttt{getBinds}\,\Gamma)\,(i@(\texttt{Identifier}\ \_)))\\ \Gamma \quad \vdash \quad f\ :\ (\texttt{getType}\, p_k);\ b\end{array}}{\Gamma\ \vdash\ (\texttt{UnionCtor}\, i\, f)\ :\ (\texttt{UnionT}\,[p_1, \ldots, p_N]);\ b}(UCtor\ :_2)$$

5.5.8 UnionCase

The first argument of a `UnionCase` expression must have a `UnionT` type, and the second one must be an identifier. The type of expression is always Boolean:

$$\frac{\Gamma\ \vdash\ f:\ (\texttt{UnionT}\ \_);\ b}{\Gamma\ \vdash\ (\texttt{UnionCase}\, f\, i@(\texttt{Identifier}\ \_))\ :\ \texttt{BooleanT};\ b}(UCase\ :)$$

5.5.9 RestrictType

The `RestrictType` expression operates on the whole collection of types which can be associated with its argument. Since we are are using single-derivation notation for our type rules, the rule for `RestrictType` just asserts that if a functional block f has, in particular, some type t, then restriction of f to any type t' which is equivalent to t succeeds. The type of the expression can be either t or t':

$$\frac{\Gamma\ \vdash\ f\ :\ t;\ b \qquad \Gamma\ \vdash\ t \equiv\ t'}{\begin{array}{l}\Gamma\ \vdash\ (\texttt{RestrictType}\, f\, t')\ :\ t;\ b\\ \Gamma\ \vdash\ (\texttt{RestrictType}\, f\, t')\ :\ t';\ b\end{array}}(RestrType\ :)$$

5.5.10 ConvertTo

As was already mentioned in Section 4.2.3, there is no uncontrolled `pretend` in Aldor--: explicit type coercions are limited to compatible types. The types are compatible if if their *ultimate representations* are equivalent, so the following type rule apply:

$$\frac{\Gamma\ \vdash\ x\ :\ t;\ b \qquad \Gamma\ \vdash\ (\texttt{ultimRep}\, t)\ \equiv\ (\texttt{ultimRep}\, t')}{\Gamma\ \vdash\ (\texttt{ConvertTo}\, x\, t')\ :\ t'; b}(ConvertTo)$$

Here the function `ultimRep` is the same as in Section 4.2.3.

5.5.11 Add1, Add2

5.5.12 With1, With2

5.5.13 VoidExpr

5.6 Values

5.7 Types

6 An overview of the Aldor-- implementation

6.1 Compilation process for the original Aldor

The original Aldor compiler was written in C, and was a proprietary software owned by IBM and NAg. The compiler is now available for free download from `http://www.aldor.org`, in the form of pre-compiled binaries for most common platforms (such as GNU/Linux on Intel x86). The Aldor compiler is still not available as Open Source software, although most of the Aldor libraries have been released as Open Source.

For a given Aldor source file (“`.as`”), the Aldor compiler, called `axiomxl`, executes the following translation steps:‘

1. The source is pre-processed: `#include` directives (similar to those used in C source files) are carried out, and macro expansion is performed.

2. The pre-processed source is parsed and converted into an intermediate in-memory representation, — the Abstract Syntax Tree (AST).

3. The AST is type-checked. However, type-checking performed by the original Aldor compiler is unnecessary restrictive. In particular, target type expressions are not evaluated, and the type-checker requires *syntactic* equivalence between the type of the left-hand side and the target (right-hand side) type.

4. The AST is converted into a platform-independent byte-code representation called FOAM (First-Order Abstract Machine). The FOAM code is similar in its purpose to Java Byte-Code. It provides an abtraction layer for compiled Aldor programs, to make them independent from the underlying hardware and the operating system.

5. The FOAM code can then be interpreted, or stored in platform-independent object libraries (“`.ao`” files) for future use, or converted into C code and then compiled into a “native” (platform-specific) binary format using a C compiler available for that platform. Note that the Aldor compiler itself does not generate any native binary code.

This explains why the original Aldor compiler cannot evaluate target type expressions during type-checking: it can only interpret FOAM byte-code which is not yet generated at the type-checking phase.

6.2 Compilation process for Aldor--

To rectify this problem, a new type-checking and evaluation engine (called `AETHER`) has been implemented for Aldor--. It is written in Haskell **??** and operates as a co-process to the original `axiomxl` compiler, communicating with the latter through external files. The executable code of the `AETHER` is generated by the Glasgow Haskell Compiler [4] (GHC), and is currently availabe for the Linux/x86 platform. The `AETHER` source code is completely platform-independent, and can be ported to any platform supported by the GHC.

The algorithm of `AETHER` is the following:

1. The `axiomxl` compiler was modified (under the permission of the copyright owners) to support a new `-Fhs` command-line option which allows the user to output the generated abstract syntax tree into an external file.

2. The `AETHER` program is then invoked. It reads the AST from a file and converts it into the hierarchy of functional blocks described in Section 3. Such a conversion is necessary because the original AST structure does not closely reflect the functional structure of Aldor-- programs, and is not very well suitable for type-checking.

3. The program is *decorated* by enumeration attributes and links: unique enumerators are given to all functional blocks, definitions and occurrencies of identifiers, and up-links are constructed between the functional blocks to provide a hierarchy of enclosing name scopes.

4. Next, each identifier within the program is bound to all its possible target(s); in other words, name bindings are constructed for the program structure in question (see Section **??**). From the implementational point of view, name bindings is a finite map from the set of indentifiers'*enumerators* to the power set of bind targets. The bind targets are uniquely identified by the enumerators of the objects they contain, e. g. definitions or functional blocks.

5. The program (i. e. the top-level FBlock) is then recursively type-checked according to the type rules presented in Section 5. Target type expressions are reduced whenever necessary. The reduction and type-checking processes are closely interleaved.

   Name bindings play the central rôle in both evaluation and type-checking operations. The bindings can only be reduced, never expanded, at this stage. Reduction of name bindings occurs when improper bindings are removed as a result of type-checking, or by run-time constraints during the reduction of target type expressions. in this way, overloaded names are resolved. For a well-typed and unambiguous program, each identifier must be bound to exactly one target at the end of the type-checking stage.

6. If type-checking succeeds, the `AETHER` co-process can interpret the Aldor-- program in question, without the need to resume the main Aldor compiler (`axiomxl`).

   The user may also want to generate the FOAM code for that program, so it can be integrated with other Aldor programs (not necesserily Aldor-- ones). The problem is, however, whether the FOAM generation component of `axiomxl` would be able to produce a valid byte code for a program which is considered ill-typed by the *original* Aldor type-checker, but was successfully type-checked by `AETHER`. This issue requires further investigation. If problems with FOAM code generation do occur, the easiest solution would be to export a modified AST back to the main translator. This new AST would contain explicit type coercion (`pretend`) nodes to make it formally acceptable for the original type-checker as well. This method has been validated in [15]: it was demonstrated that using the `pretend` expressions in standard Aldor allows us to by-pass the limitations of its type-checker and still generate valid FOAM and executable programs.

7 Applications of Aldor--

This section introduces a variety of examples in Aldor--. We first revisit the examples given by way of motivation in [15] and then, building on these examples, we give a logical derivation in the theory of monoids.

7.1 Logical examples, revisited

The paper [15] contains a variety of logical examples, written in Aldor. Here we show how these examples can be rewritten in Aldor--, and point to the differences that result.

Implication

Implication is represented by the function type, since a proof of an implication `(A=>B)` can be thought of as a transformation that turns a proof of `A` (its input) into a proof of `B` (its output). A proof of an implication is introduced by forming a function; a proof of an implication `(A=>B)` is eliminated by applying it to a proof of `A` to yield a proof of `B`; this is the *modus ponens* rule. An example is given in Figure 2.

```
-- The example ((A=>(B=>C))=>(A=>B)=>(A=>C))

S (A:Type, B:Type, C:Type, p: ((a:A) -> ((b:B)->C)) ) :
  ((q: ((a:A)->B) ) -> ((a:A)->C ) )
  ==
   (q: ((a:A)->B))  :  ((a:A)->C) +-> ((a:A):C +-> ((p a)(q a)));
```

Figure 2: An example using implication: the S combinator

```
-- The conjunction type, And.

And (A: Type, B: Type): Type ==
    add
    {    Rep == Record(fst:A,snd:B);

        andIntro(a:A,b:B):% == per [a,b];
        andElim1(p:%):A     == (rep p).fst;
        andElim2(p:%):B     == (rep p).snd; };

-- An example proof that A&B => B&A

flip(A:Type,B:Type,p:And(A,B)):And(B,A) ==
{    X: Type == And (A,B); Y: Type == And (B,A);

    (andIntro $ Y) ((andElim2 $ X) (p), (andElim1 $ X) (p)); };
```

Figure 3: Aldor-- conjunction and an example proof

Conjunction

The conjunction operation, `And`, is introduced in Figure 3 as a function over types: it has two arguments of type `Type` and it returns a result of type `Type`. The result is an *abstract data type*, whose representation is a record with fields `fst` and `snd` of type `A` and `B`, respectively. Logically this corresponds to the fact that a proof of `A&B` is given by a proof of `A` together with a proof of `B`.

Introduction and elimination operations construct and destruct a record, modulo conversions between the ADT and the concrete, carrier, type given by the functions `rep` and `per`.

In the example proof in Figure 3 the *qualification* operation `$` is used to qualify the operations `andIntro`, `andElim1` and `andElim2`. `$` forms a *qualified name*, by way of disambiguating the use of an (unqualified) identifier, as in Aldor itself.[6] Note that `X` and `Y` are used as placeholders for qualifiers for the `Add1` operation; a qualifier is required to be an `Identifier`, rather than an arbitrary expression. In the particular case here the disambiguation has the effect of choosing the particular instance at which to use a parameterised ADT: conjunction elimination is used over `And(A,B)`, whereas introduction is used to introduce an element of `And(B,A)`.

In Figure 4 conjunction is presented in an unencapsulated form. The type `AND(A,B)` actually *is* the type `Record(fst:A,snd:B)`, and the introduction and elimination functions are operations over records,

[6]Disambiguation using `$` is discussed in Section 8.3 of the Aldor User Guide [16].

```
-- The conjunction type, AND, unencapsulated.

AND (A:Type,B:Type): Type == Record(fst:A,snd:B);

ANDIntro(A:Type,B:Type,a:A,b:B):AND(A,B) == [a,b];
ANDElim1(A:Type,B:Type,p:AND(A,B)):A     == p.fst;
ANDElim2(A:Type,B:Type,p:AND(A,B)):B     == p.snd;

-- An example proof that A&B => B&A

flip(A:Type,B:Type,p:AND(A,B)):AND(B,A) ==
    ANDIntro(B,A,ANDElim2(A,B,p),ANDElim1(A,B,p));
```

Figure 4: Unencapsulated conjunction and an example proof in Aldor--

unmediated by `rep` and `per`. For this approach to work in Aldor-- it is necessary to have `Type` expressions evaluated: for example, for the definition of `ANDIntro` to typecheck it is necessary for the type `AND(A,B)` to evaluate to the record type `Record(fst:A,snd:B)`.

In the unencapsulated case the particular type at which an introduction or elimination rule is applied is determined by the *explicit* type parameters which constitute the first two arguments of each of these functions. Contrast this with the encapsulated treatment, where the qualification operation, `$`, is used to select the type at which the operations are used. In a language with parametric polymorphism, this information would be supplied implicitly, but the polymorphism in Aldor--is explicit, given by the `Type` parameters.

Note that in Aldor itself, the unencapsulated implementation requires explicit type conversion, using `pretend`, from the type expression `AND(A,B)` to its defined value as a record type.

Disjunction

A disjunction is represented by a union type: a proof of `A|B` is, constructively, either a proof of `A` or a proof of `B`. A proof of a disjunction is constructed by injecting the proof of the corresponding component part into the union type. A disjunction is eliminated by *proof by cases*: to prove `C` from `A|B` it is necessary to prove it, separately, from `A` and from `B`.

Two implementations of disjunction are given. Figure 5 implements it as an ADT, and gives a proof of the example `((A|B)=>C) => (A=>C)&(B=>C)`. The same example is proved in the unencapsulated treatement given in Figure 6; the structure of the two proofs is similar, and the contrast between the two echoes the `flip` example. In the encapsulated case, qualification (using `$`) is used to identify the (type-)correct instance of the various functions; in the unencapsulated case, explicit type arguments supply the same information.

Negation

The constructive intrepretation of negation is defined using an *absurd* proposition: that is, a proposition that has no proof, or equivalently, a type that has no elements. This is the Aldor-- type `Exit`. The negation of `A` is then defined to be

```
(a:A)-> Exit
```

```
-- The disjunction type, Or

Or(A:Type, B:Type) : Type ==
    add {   Rep == Union(inl:A, inr:B);

            orIntro1(a:A):% == per (union (inl==a));
            orIntro2(b:B):% == per (union (inr==b));
            orElim (C:Type, f: (a:A)->C, g: (b:B)->C, p:%) : C
            ==
            {   val: Union(inl:A,inr:B) == (rep p);

                if (val case inl) then f(val.inl) else g(val.inr) };
         };

-- The example that ((A|B)=>C) => (A=>C)&(B=>C)

andOr(A:Type, B:Type, C:Type, p: (x: Or(A,B)) -> C) :
     And((a:A)->C, (b:B) ->C)
     ==
     {  X: Type == And ((a:A)->C, (b:B) ->C);
        Y: Type == Or  (A, B);

         (andIntro $ X)
            ((a:A):C +-> p ((orIntro1 $ Y) (a)),
             (b:B):C +-> p ((orIntro2 $ Y) (b)))
    };
```

Figure 5: Aldor-- disjunction and an example proof

```
-- The unencapsulated disjunction type, OR

OR(A:Type, B:Type) : Type == Union(inl:A, inr:B);

ORIntro1(A:Type,B:Type,a:A):OR(A,B) == union (inl==a);
ORIntro2(A:Type,B:Type,b:B):OR(A,B) == union (inr==b);
ORElim(A:Type,B:Type,C:Type, f: (a:A)->C, g: (b:B)->C, p:OR(A,B)) : C
  == if (p case inl) then f(p.inl) else g(p.inr) ;
         };

-- The example that ((A|B)=>C) => (A=>C)&(B=>C)

andOr(A:Type, B:Type, C:Type, p: (x: OR(A,B)) -> C) :
     AND((a:A)->C, (b:B) ->C)
     == ANDIntro((a:A)->C,
                 (b:B)->C,
                 (a:A):C +-> p (ORIntro1(A,B,a)),
                 (b:B):C +-> p (ORIntro2(A,B,b)));
```

Figure 6: Unencapsulated disjunction and an example proof

From a proofs of `A` and its negation it is then possible to derive a proof of `Exit`, which in turn leads to a proof of *any* formula, through `exfalso`. Encapsulated and unencapsulated variants of negation are presented in Figures 7 and 8.

Adding the law of the excluded middle (LEM) makes the logic classical. The proof in Figure 9 shows that the law of double negation elimination follows from LEM. We leave the proof of this in the unencapsulated version as an exercise for the reader.

7.2 Equality

We treat equality axiomatically, by introducing operations forming both types and values *without definition bodies*; their definitions simply declare their types, and type checking will ensure that they are only used in type-correct ways. `Eq` is the type constructor, and `eqRefl`, `eqSymm` and `eqTrans` are the three axioms for an equivalence relation.

The version of `eqSubst` presented here is a special case of a general substitution mechanism:[7]

```
leibnitz: (T: Type, a: T, b: T, e: Eq (T, a, b), P:(x:T)->Type, p: P(a)) -> P(b)
```

by taking `P` to be

```
(x:T): ... -> Eq(T1, patt(a), patt(x))
```

where `P(a)` will be `Eq(T1, patt(a), patt(a))`, which is proved by the reflexivity axiom, `eqRefl`.

```
-- The negation type: Not

exfalso : (e: Exit, B: Type) -> B;  -- No body; this function returns ERROR

Not(A:Type) : Type
    == add
    {   Rep == (a:A)-> Exit;
        notIntro (p: (a:A) -> Exit) : % == per p;
        notElim  (p:%, q:A):  Exit == (rep p) q ;

        contraRule: (B:Type, p:%, q:A) -> B
        == exfalso (notElim (p,q), B);
};

-- Example ((A=>B)&(A=>~B))=>(A=>C)

contraAx (A:Type, B:Type, C:Type, p: And((a:A)->B, (a:A)->Not(B))) : ((a:A)->C)
    ==
    {   X: Type == And((a:A)->B, (a:A)->Not(B));

        (a:A):C +->
        contraRule(C,
                  ((andElim2 $ X) p) a,
                  ((andElim1 $ X) p) a);
    }
```

Figure 7: Aldor-- negation and an example

```
-- The unencapsulated negation type: NOT

exfalso : (e: Exit, B: Type) -> B;  -- No body; this function returns ERROR

NOT(A:Type) : Type == (a:A)-> Exit;

notIntro (A:Type, p: (a:A) -> Exit) : NOT(A) == p;
notElim  (A:Type, p:NOT(A), q:A):  Exit      == p q ;

contraRule: (A:Type, B:Type, p:NOT(A), q:A) -> B
        == exfalso (notElim (A,p,q), B);

-- Example ((A=>B)&(A=>~B))=>(A=>C)

contraAx (A:Type, B:Type, C:Type, p: AND((a:A)->B, (a:A)->NOT(B))) : ((a:A)->C)
    ==
    {   X: Type == (a:A)->B;          -- used below as
        Y: Type == (a:A)->NOT(B));    -- abbreviations

        (a:A):C +->
        contraRule(B,C,
                  (ANDElim2(X,Y,p)(a)),
                  (ANDElim1(X,Y,p)(a)) );
    }
```

Figure 8: Aldor-- negation and an example, unencapsulated

```
-- Example ((A|~A)&~~A => A)

notNot (A:Type, p:Or(A,Not(A)), q:Not(Not(A))): A ==
{   N: Type == Not (A);
    NN:Type == Not (Not (A));
    X: Type == Or (A,N);

    (orElim $ X) (A,
                 (a:A):A +-> a,
                 (r: Not(A)):A +-> (contraRule $ NN) (A,q,r),
                 p);            };
```

Figure 9: Double negation elimation follows from the law of the excluded middle

```
-- The Equality Type Axioms:

#include "basiclib"

Eq:      (T: Type, a: T, b: T) -> Type;

eqRefl:  (T: Type, a: T) -> Eq (T, a, a);

eqSymm:  (T: Type, a: T, b: T, e: Eq (T, a, b)) -> Eq (T, b, a);

eqTrans: (T: Type, a: T, b: T, c: T,  e1: Eq (T, a, b), e2: Eq (T, b, c)) ->
         Eq (T, a, c);

eqSubst: (T: Type, a: T, b: T, e: Eq (T, a, b), T1:Type, patt: (z: T) -> T1) ->
         Eq (T1, patt(a), patt(b));
```

Figure 10: The axioms for equality

```
MonoidGen: Category == with
{
  mult: (a: %, b: %) -> %;
  I: %;

  -- Monoid axioms:

  leftUnit:  (a: %) -> Eq (%, a, mult(a,I));
  rightUnit: (a: %) -> Eq (%, a, mult(I,a));
  assoc:     (a: %, b: %, c: %) ->
                 Eq (%, mult(a, mult (b,c)), mult(mult (a,b), c));
}
```

Figure 11: Axiomatising the Monoid category

```
MonoidO2: Category == MonoidGen with
{
  order2: (a: %) -> Eq (%, (mult$MonoidGen) (a,a), (I$MonoidGen));
}

comm2 (M2: MonoidO2, a: M2, b: M2):
         Eq (M2, (mult$M2) (b,a),  (mult$M2) (a,b)) == ...
```

Figure 12: The `MonoidO2` category and the communtativity assertion

7.3 Axiomatising the monoid category

Figure 11 contains the category of monoids, axiomatised, as first discussed in [15]. Note that we cannot use the infix operator ‘*’ for multiplication here, since ‘*’ is by default left-associative, so the Aldor parser would re-combine exprs involving ‘*’ in a way we don’t want. We therefore use `mult` instead.

7.4 An algebraic proof

One of the first non-trivial results of the theory of monoids is that a monoid in which every square is equal to the identity is in fact abelian. In our proof we show that in such a monoid, axiomatised by `MonoidO2` in Figure 12, has the required property by defining the value `comm2`, the commutativity assertion. The value `comm2` is the final value in a series of proofs, given by

The full proof `CommMonoid.as` can be found on the project web page.

8 Conclusions and Future Work

We would like to acknowledge the assistance of Stefan Kahrs and Claus Reinke for discussions about overloading.

References

[1] Alfa. Available from `http://www.math.chalmers.se/~hallgren/Alfa/`.

[2] Giuseppe Castagna and Gang Chen. Dependent types with subtyping and late-bound overloading. *Information and Computation*, 168(1), 2001.

[3] Nigel J. Cutland. *Computability*. Cambridge University Press, 1981.

[4] The Glasgow Haskell Compiler. Available from `http://www.dcs.glasgow.ac.uk/fp/software/ghc/`, 1998.

[5] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Principles of Programming Languages 22*, pages 130–141, San Francisco, CA, January 1995.

[7]Named after the mathematician Gottfried Wilhelm Leibnitz (1646 - 1716), who coined the principle of ‘substitution of equals for equals’.

```
(b*a)*(b*a)      = I
(b*a)*(b*a)      = ((b*a)*b)*a
((b*a)*b)*a      = (b*a)*(b*a)
((b*a)*b)*a      = I
(((b*a)*b)*a)*a = I*a
a                = I*a
I*a              = a
(((b*a)*b)*a)*a = a
((b*a)*b)*(a*a) = (((b*a)*b)*a)*a
a*a              = I
I                = a*a
(b*a)*b          = ((b*a)*b)*I
((b*a)*b)*I      = ((b*a)*b)*(a*a)
(b*a)*b          = ((b*a)*b)*(a*a)
(b*a)*b          = (((b*a)*b)*a)*a
(b*a)*b          = a
((b*a)*b)*b      = a*b
(b*a)*(b*b)      = ((b*a)*b)*b
b*b              = I
I                = b*b
b*a              = (b*a)*I
(b*a)*I          = (b*a)*(b*b)
b*a              = (b*a)*(b*b)
b*a              = ((b*a)*b)*b
b*a              = a*b
```

Figure 13: Proving comm2

[6] Ralf Hinze and Johann Jeuring. Generic Haskell: Practice and Theory. In *Lecture Notes of the Summer School in Generic Programming*, 2002/2003.

[7] John Hughes and Simon Peyton Jones, editors. *Report on the Programming Language Haskell 98*. `http://www.haskell.org/report/`, 1999.

[8] Richard D. Jenks and Robert S. Sutor. *Axiom: The Scientific Computation System*. Springer, 1992.

[9] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.

[10] The LEGO Proof Assistant. Available from `http://www.dcs.ed.ac.uk/home/lego`.

[11] Erik Poll and Simon Thompson. Integrating Computer Algebra and Reasoning through the Type System of Aldor. In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems, 2000*. LNCS 1794, Springer-Verlag, 2000.

[12] B. Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994.

[13] The Coq Development Team. The Coq Proof Assistant Reference Manual, Version 7.3. Technical report, INRIA, 2002. Available from `http://pauillac.inria.fr/coq/doc/main.html`.

[14] Simon Thompson. *Type Theory and Functional Programming*. International Computer Science Series. Addison-Wesley, 1991.

[15] Simon Thompson. Logic and Dependent Types in the Aldor Computer Algebra System. In Manfred Kerber and Michael Kohlhase, editors, *Symbolic Computation and Automated Reasoning: the Calculemus 2000 Symposium*. A. K. Peters, 2001.

[16] S.M. Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, J.M. Steinbach S.C. Morrison, and R.S. Sutor. *AXIOM Library Compiler User Guide*. The Numerical Algorithms Group (NAG) Ltd., 1994.