The Type System of Aldor

Erik Poll and Simon Thompson

July 1999

Abstract

This paper gives a formal description of – at least a part of – the type system of Aldor, the extension language of the computer algebra system AXIOM. In the process of doing this a critique of the design of the system emerges.

Contents

1 Introduction

The computer algebra system AXIOM and its precursor Scratchpad are unusual among computer algebra systems in that they are strongly typed, so that any errors in the types of expressions or programs are caught prior to the programs being executed. In order to give types to the basic objects of mathematics it is necessary to develop an expressive and flexible system of types. In this paper we examine the programming language Aldor[WBD$^+$94], which has grown out of AXIOM as a compiled 'extension language',[1] that is a language in which to write libraries for AXIOM or indeed other computer algebra systems such as Maple.

One might argue that most users of computer algebra systems prefer the freedom provided by an interpreted and weakly typed language. While this is the case for small-scale experimental use, a compiled language promises users efficiency, whilst as it is strongly typed it can assure users that their programs are free of potential type errors; both these properties are desirable for library code which will in general be substantial and executed repeatedly.

As was said earlier, Aldor has a very complicated and expressive type system in order to render the types appearing in a computer implementation of basic mathematics. This requirement represents a substantial challenge, and it is interesting for instance to observe that the motivating example for an extension of the C++ type system described in [BR95] comes from computer algebra. Among other things, the type system of Aldor includes so-called dependent types, types as values, a rich system for abstract datatypes – provided by so-called domains and categories – and overloading. These and other features of the Aldor type system are discussed in section 2.

The Aldor User Guide [WBD$^+$94] gives an informal description of the Aldor type system. In this paper we will try to give a *formal* description of at least part of the type system of Aldor. A formal description gives a clear and unambiguous description of the types of the language; this can provide a foundation for the

[1] In the past Aldor has also been known as AXIOM-XL and A$^\sharp$.

implementor as well as allowing general properties of the type system to be studied.

This formal description is a first step of the project described in [PT98], namely of incorporating a logic into Aldor. This is done by using the so-called 'propositions as types' or Curry-Howard correspondence, [How80], under which logical propositions are encoded as types of a functional programming language. This encoding will be made possible in Aldor by making a modification to allow type expressions as well as ordinary expressions to be evaluated.

The formal description will be given by a typing relation of the form $\Gamma \vdash t : T$ which is read as "in context Γ term t has type T"; this relationship will be defined by a set of type inference rules.

It is important to realise that – lacking a good description of the type system – the basis for our formal description is the compiler itself. Any questions about the type system have been resolved by experimenting with simple programs to see which ones are rejected by the compiler because of typing errors. Of course, this has its limitations.

Two central questions that arise are

- *How much of Aldor do we try to formalise?*
  We do not attempt to give a formal description of the entire type system of Aldor. Instead, we only describe what we consider to be the "core" of the Aldor type system, which includes the essential features but excludes some of the more baroque ones. The latter may be features that we want to exclude because they do not seem interesting (they can be seen as 'syntactic sugar', for instance) or are too *ad hoc*, or features that we have to disregard in order to keep things simple enough to formalise. Ideally, the core of the Aldor type system we describe should be a "small" type system, in the sense that it is built by combining of a small number of orthogonal primitives for constructing types.

  It will always remains a point of discussion whether we should include more or less of the Aldor type system in the formal description. At several places we will point out constructions that are possible in Aldor which we have not included in our formal description and we collect together a list of these in Section 16.

- *Does the formalisation describe Aldor as it is, or as we'd want it to be?*
  There are several cases where the type system of Aldor (or rather, the implementation of the type system in the compiler) behaves strangely. Here one can ask if, instead of giving very complex rules that exactly describe this, it would not be more useful to propose simpler, more sensible, typing rules that result in a "cleaner" type system. At several places we will point out where our formalisation does not accurately describe the behaviour of the Aldor compiler; we collect these points in Section 18.

In experimenting with simple programs we came across a number of bugs in the compiler, and across cases where the compiler behaved strangely, either accepting seemingly ill-typed programs or rejecting seemingly well-typed ones.

1.1 Related Work

There has been a lot of interest in programming languages with types-as-values in the 1980's, see, for instance, [DD85, MR86, LB88]. Recently there has been renewed interest in languages with dependent types, as evidenced by [Aug98, DTP99].

There has been a lot of work in type theory that is relevant here. Related to the dependent types of Aldor is the work on so-called constructive type theories, such as the constructive type theories of Martin-Löf [ML79] or the Calculus of Constructions [CH88]. One useful notion here is that of Pure Type System (PTS) [Bar93], which provides a general framework for giving compact characterizations of many type systems with function types and dependent types, and makes it easy to compare such systems.

Related to the module system of Aldor (as provided by its domains and categories) is the work on different variants of "sum types" for describing modules e.g. in the setting of the functional programming languages Standard ML (SML) [MTHM97, Mac86, Rus98].

More closely related to Aldor itself, [San95] proposes a type system for computer algebra which is based on Aldor. The focus of Santas' paper is on the module system. The type system described does not include type-as-values or dependent types. Finally, the type system of Aldor has been investigated using the categorical notion of a sketch, [Tou98].

Acknowledgements

We are grateful to NAG, and Mike Dewar in particular, for granting us access to the Aldor source code, Version 1.1.10b. Martin Dunstan has helped us to understand some of the intricate details of the internals of the Aldor source code, and Chris Ryder's work [Ryd98] on understanding the mechanics of type checking in Aldor was most useful. Stephen Watt answered a number of queries about typing in Aldor as well as listening patiently to our ideas about how it might be modified.

2 Introduction to the Aldor type system

Before giving a formal description of the type system, this section gives an informal introduction of the main features of the Aldor type system and illustrates these with some simple examples. Subsequent sections will give a more detailed explanation of these features.

Aldor is not a functional language, but an imperative one. However, Aldor does have a complete functional language as a sub-language (which, for instance, includes higher-order functions). In the formal description here we will limit ourselves to this functional sub-language of Aldor, i.e. we disregard any of the imperative features of Aldor. The functional sub-language of Aldor does in fact contain all the interesting type constructions of Aldor; one can view the imperative features in a similar way to those of SML, with the proviso that SML's type system is made more complex by the interaction of reference types and parametric polymorphism.

Aldor provides many of types familiar from other programming languages, such as function types, product types, record types, union types and so on, with the usual terms of these types: functions, products, records, for instance. For example, the fragment of Aldor program below defines a function `double` and a record `rr`:

```
double : Integer -> Integer
       == (n:Integer) : Integer +-> n+n ;

rr : Record (i:Integer, j:Boolean)
   == [i==4,j==true];
```

But, in Aldor these familiar constructs can be more complicated than in most other languages. This is mainly due to the two of the features discussed below: *dependent types* and *types as values*. These and other aspects of the language are examined informally now.

Dependent Types

Aldor allows so-called dependent types. One of the standard examples of a dependent type is the type `Vector(n)` of, say, floating point vectors of length `n`. This is called a *dependent* type, because it depends on the – in this case, integer – value `n`.

Functions can have dependent types, in which the type of a function result depends upon the value of a parameter. An example is a function

```
vectorSum : (n:Integer) -> Vector(n) -> Float
```

which takes as arguments an integer `n` and a vector of type `Vector(n)`, i.e. a vector of length `n`, and returns the sum of that vector. The result of a function application, say

```
vectorSum(34)
```

has the type `Vector(34) -> Float` because its argument has the value `34`.

Another example of a function with a dependent type is the append function for vectors:

```
append : (n:Integer,m:Integer,Vector(n),Vector(m)) -> Vector(n+m)
```

There are two important points about dependent types: first, following the Curry-Howard isomorphism – better known as "propositions as types" – a type system with dependent types is powerful enough to express predicates with universal quantification [How80]. Dependent types are commonly used in this way in so-called constructive type theories, such as Martin-Löf's Type Theory [ML79, Tho91] or the Calculus of Constructions [CH88]. Second, there is a well-known price to be paid for dependent types (see [MR86, Aug98] for instance), namely that type checking of programs will involve executing parts of programs. This will be discussed in Section 14.

The Aldor type system contains a second form of type dependence, in this case between the fields of records. As an example consider

```
rec : Record (n:Integer, v:Vector(n))
  == [ n==3, v==vec3 ];
```

which defines a record containing two fields; the first, `n`, is an integer, whilst the second is a vector whose length is `n`. These types can express predicates with existential quantification

Dependent functions and records support universal and existential quantification, and so it should be possible to represent any proposition of first-order logic by means of an Aldor type. This is not possible in the current implementation since there is no evaluation of type expressions, so that, for example, the types `Vec(5)` and `Vec(2+3)` are seen as different types. Our aim, discussed in [PT98], is to rectify this anomaly. type system should be powerful enought to represent an

Types as values

Most programming languages enforce a strict separation between a collection of terms – or values – and a collection of types. But Aldor treats types as terms like any other: a type such as `Integer->Integer` can be manipulated in the same way as any ordinary expression like `3+4`. In particular,

- Just as other terms have types, so do the types themselves: there is a special constant `Type` that is "the type of all types". For example, `Boolean` : `Type` and indeed `Type` : `Type`.
- Any construction that is possible with terms is also possible with types. This means that types can be passed as arguments to a function, or returned as the result of a function. For example, the function

  ```
  List : Type -> Type
  ```

  takes a type as input and produces a type as output. Applying the function `List` to the type `Integer` produces a type `List(Integer)`, the type of lists of integers.

  Type can also be used as components of records. For example, the record

  ```
  tt : Record (t:Type, b:Boolean)
          == [t==Integer,b==true];
  ```

  has a `field` whose value is a type.

Of a language such as Aldor it is often said that types are treated as "first-class citizens" (as opposed to most other languages, where types are only second-class citizens).

To fully exploit the idea of types-as-values dependent types are effectively indispensable. Combining types-as-values and dependent types, we can make the polymorphic (or generic) functions that exists in functional programming languages like ML or Haskell. For example, a polymorphic function `reverse` that reverses a list with elements of an arbitrary type could be typed as follows

```
reverse : (T:Type) List(T) -> List(T)
```

A difference with functional programming languages like ML or Haskell is that in Aldor such polymorphic functions like `reverse` have to be given explicit type parameters, whereas in modern functional languages these type parameters are inferred by the compiler, using so-called Hindley-Milner type inference [Mil78].

The fact that types can be used as values greatly increases the expressive power of the language. But, as mentioned before, there is a price to be paid for the associated dependent types (see e.g. [MR86]).

Domains and Categories.

Aldor provides a rich system for abstract datatypes, called *domains*, and for the types of datatypes called *categories*. Categories effectively describe the interface or signatures of abstract datatypes. The domains and categories of Aldor make it possible to model the rich universe of mathematical structures that arise in computer algebra, e.g. of rings, fields, etc., as well as the relationships between them, e.g. every field is also a ring.

An example of a category is

```
Ring : Category == with {+ : (%,%) -> %;
                         * : (%,%) -> %;
                         1 : %;
                         0 : % }
```

which describes the interface of rings, i.e. the operations that any type `%` has to provide in order for it to be a ring. It is then possible in Aldor to write so-called *generic* (or polymorphic) algorithms, e.g. a summation algorithm that works for arbitrary rings:

```
sum : (R:Ring) List(R) -> R
```

Note that this provides a further example of a dependent type in use: the type of the result of applying `sum` to `R`, namely `List(R) -> R`, depends on the ring `R`.

Overloading

Aldor allow overloading, so that the same name can be used more than once, provided any resulting ambiguity can be resolved by the type system. So the same name can only be used to refer to terms of different types. The standard example of overloading is the use of `+` as a binary operator for different types, e.g. both `+:(Integer,Integer)->Integer` and `+:(Real,Real)->Real`.

Subtyping

Aldor provides a form of subtyping. The most interesting source of subtyping are the categories, where subtyping captures the notion of an interface being subsumed by a richer interface

For example, the category `Monoid`

```
Monoid : Category == with {* : (%,%) -> % }
```

is a supertype of `Ring`, capturing the intuitive idea that every ring is also a multiplicative monoid. This means that a ring can be used in any context where a monoid is expected.

In fact, Aldor distinguishes three forms of subtyping: in addition to "subtyping" between types, there are also so-called "courtesy conversions" between types and there is also a notion of "type satisfaction". These will be discussed in Section 15.

Multiple Values

Finally, one of the more puzzling features of the Aldor type system is the notion of multiple value. A multiple value is essentially a sequence of terms $(t_1, \ldots, t_n)$, which are very similar to n-ary products, or cross products in Aldor terminology. Indeed, the notation of a multiple value and a cross product is exactly the same, and there exist courtesy conversions (see Section 15) from multiple values to cross products and back. It is not clear to us why Aldor provides both multiple values and cross products. We have left out multiple values from the formal description of Aldor given here.

3 The formal description of the Aldor type system

As mentioned in the previous section, in the formal description of the Aldor type system we ignore all imperative features of Aldor, and only describe a purely functional sub-language of Aldor. So we do not consider the statements of Aldor, e.g. assignments, for-loops, etc. In particular this means that whenever we talk about "variables" these are never variables in the sense of imperative programming – i.e. memory locations – but always variables in the sense of "formal parameters".

3.1 Typing relations

The typing relation is formally described by typing judgements of the form

$$\Gamma \vdash t : T.$$

The judgement $\Gamma \vdash t : T$ is read as "term t has type T in context Γ". Here the context Γ is the list of all the variable declarations, type definitions, etc., that are in scope. Simple examples of typing judgements are:

$$\begin{array}{lll} \Gamma & \vdash & \texttt{true} : \texttt{Boolean} \\ \Gamma & \vdash & + : (\texttt{Integer}, \texttt{Integer})\texttt{->Integer} \end{array}$$

If $\Gamma \vdash t : T$ then we say that t and T are *well-formed* expressions (a well-formed term and type expression, respectively) in context Γ. To define $\Gamma \vdash t : T$ we need an auxiliary judgement $\Gamma \vdash ok$, which means "context Γ is well-formed".

Remark 3.1 (Terminology) Our terminology is different from that used in the Aldor User Guide [WBD+94]: what we call "terms" are called "values" there.

We use "terms" rather than "values" because "values" is often reserved for those expressions that are "evaluated" in some sense. For example, `3+4` and `7` are both terms, but usually only `7` is considered to be a value.

A similar distinction can be applied to terms which represent types. □

3.2 The Aldor universe

In our discussion of Aldor we will distinguish six kinds of expressions:

- *terms*,
- *types*,
- *domains*,
- *categories*,
- *abstract datatypes (ADT's)*,
- *packages*.

Below we explain their intuitive meanings and the basic relations between them.

The coarsest distinction between different kinds of expressions one can make in Aldor is between *terms* and *types*. But, since types are values, and the types themselves also are terms, so that *types* $\subseteq$ *terms*. And the type of all types – `Type` – is itself a type, so `Type` $\in$ *types*. Similarly, the type of all categories – `Category` – is a type, so `Category` $\in$ *type*.

We distinguish the following subsets of *terms* and *types*, called *domains* and *categories*, that are of special interest:

- *domains* $\subseteq$ *terms*.
  Domains can either be *abstract datatypes* (ADT's) or *packages*.

  Packages are collections of definitions, which can include definitions of functions, types, or any other terms. These definitions are called the exports of a package. We can think of packages as libraries and also, by analogy with SML, as *structures*.

  Like packages, ADT's are collections of definitions, but an ADT includes a distinguished definition of a type. The other definitions will typically be operations on that type. To take the standard example, an ADT `Stack` for stacks would define a representation type for stacks, and implementation of the stack operations for that particular representation.

- *categories* $\subseteq$ *types*.
  Categories are the types of domains. Basically a category describes the *interface* of a domain, i.e. it lists the exports with their types, like the example of the category `Ring` on page 6. Again by analogy with SML, categories are like SML signatures.

  Just as there is a type of all types, there is a type of all categories, which is called `Category`.

The domains that are abstract datatypes play an important role in Aldor. Although strictly speaking these ADT's are terms and not types, types are introduced when ADTs are *named*. (Aldor type naming is done in exactly the same way as for any other value; it is therefore quite possible to introduce 'anonymous' ADTs, even if they are only of curiosity value.)

For example, if we have the abstract datatype `Stack` mentioned above, then the name `Stack` is then not only used to refer to this whole collection of definitions that make up the ADT, but is also used as the name of the (abstract) type introduced by the ADT. The fact that the name of an domain is used as a type means that there is an implicit projection by means of naming from abstract datatypes to types, indicated by the dotted arrow in Figure 1.

All this leads to the view of the Aldor universe given in Figure 1.

3.3 A grammar for a subset of Aldor

The grammar given in Figure 2 defines some of the raw syntax of Aldor terms. To define the set of raw terms *Term* it also defines a set of type tuples *TypeTuple*.

There are two points to note about the grammar given in Figure 2.

- The square brackets [...] are not part of the syntax, but indicate an optional inclusion. E.g. packages can be of the form `add`$\{x_1 : T_1$`==`$t_1; \ldots; x_n : T_n$`==`$t_n\}$, of the form `add`$\{x_1$`==`$t_1; \ldots; x_n$`==`$t_n\}$, or any combination of the two.

- This distinction between terms t and types T is not a distinction that can be made formal at this stage. To tell which terms are types we have to refer to the typing relation: a term T is a type (in a context Γ) if and only if $\vdash T : Type$ (or in a context, $\Gamma \vdash T : Type$). Still, it is useful to suggest

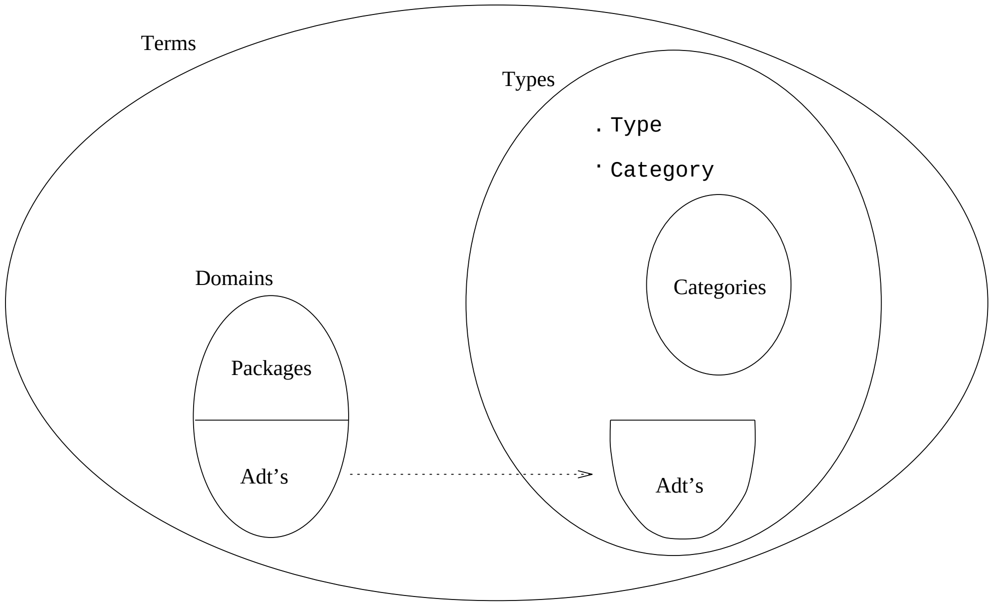

Figure 1: The Aldor universe

the distinction between terms and types already here. Throughout this report we will stick to the convention that lowercase letters range over arbitrary terms and uppercase letters range over types (or, in some cases, names for types).

Remark 3.2 (domain vs (abstract data)type) The Aldor User Guide [WBD+94] is sometimes a bit sloppy in its use of the terminology, particularly when it comes to domains and (abstract data)types.

- *domain vs abstract datatype.* The notions of abstract datatype and domain are often identified, although a domain can also be a package. More often that not "domain" should be taken to mean "abstract datatype".
- *domain vs type.* A more serious cause of confusion is that "domain" and "type" are often treated as synonyms.

  All user defined types in Aldor will typically be introduced by means of abstract datatype definitions. One can go even further and take the viewpoint that all the primitives types provided by Aldor are also abstract data types, with the difference that the definition of the primitive types cannot be given inside the language itself. In this view all types originate from abstract datatypes, so that the set of (names of) abstract datatypes is isomorphic to the set of types. Indeed, all primitive types are declared as abstract datatypes in the library file .../lib/libaxllib/lang.as that provides an interface for all language-defined types.

  This seems to explain why in the User Guide the terms 'type' and 'domain' are almost used interchangeably. (Strictly speaking it is only the domains that are abstract datatypes than can be viewed as types, but we already pointed out above that the terms domain and abstract datatype are often treated as if they were synonyms.)

$t, T \in Term$		
::=	x	variable
	$\mid (x_1 : T_1, \ldots, x_n : T_n) : T \texttt{+->} t$	abstraction
	$\mid t_1(t_2)$	application
	$\mid (t_1, \ldots, t_n)$	multiple value, or tuple, or cross product
	$\mid \texttt{record}$	record
	$\mid \texttt{union}$	union
	$\mid \texttt{bracket} \mid \texttt{explode} \mid \texttt{apply} \mid \texttt{case}$	operations on records/unions
	$\mid \texttt{Type}$	the type of all types
	$\mid \texttt{Category}$	the type of all categories
	$\mid \overrightarrow{T}_1 \texttt{->} \overrightarrow{T}_2$	function type
	$\mid \texttt{Cross}\overrightarrow{T}$	cross product type
	$\mid \texttt{Record}\overrightarrow{T}$	record type
	$\mid \texttt{Union}\overrightarrow{T}$	union type
	$\mid {}'x_1, \ldots, x_n{}'$	enumeration type
	$\mid \texttt{Tuple}\ T$	tuple type
	$\mid \texttt{add}\{x_1[: T_1]\texttt{==}t_1; \ldots; x_n[: T_n]\texttt{==}t_n\}$	package
	$\mid \texttt{add}\{\texttt{Rep==>}T;$ $x_1[: T_1]\texttt{==}t_1; \ldots; x_n[: T_n]\texttt{==}t_n\}$	ADT
	$\mid x_i \$ x$	projection from package/ADT
	$\mid \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\}$	category
$\overrightarrow{T} \in TypleTuple$		
::=	$([x_1 :]T_1, \ldots, [x_n :]T_n)$	type tuple

Figure 2: A grammar for a subset of the Aldor terms

The remainder of this report

The sections that follow give the formal description of the type system for each individual language construct of Aldor. These sections roughly follow the same format: we give the raw syntax, specified by a piece of context-free grammar, the typing rules, which impose restrictions on the raw syntax to yield the "well-types" syntax, and some examples. We will discuss the typing rules to point out any peculiarities, to point out any differences between our formal description and Aldor as it is actually implemented by the compiler, and to suggest possible improvements or simplifications of the typing rules (which would however increase the difference between our formal description and Aldor as it is implemented by the compiler).

4 Contexts

Typing depends on a context containing *declarations* and *definitions*. A context defines the set of names that are currently in scope. Contexts may include declarations of variables[2] like

```
x : Integer
```

and definitions of constants, such as

```
x : Integer == 5
```

Note that because types are values, a declaration of a variable can be a declaration a *type*-variable, e.g.

```
X : Type
```

and a definition of a constant can be the definition of a *type*-constant e.g.

```
TT : Type == Integer -> Integer
```

Definitions can be of two forms, namely of the form $x : T$==t, i.e. with an explicit type, or of the form x==t, i.e. without an explicit type.

Aldor also allows a definition to be preceded by the keyword `define`, which is intended to make the value of a definition 'visible' to its context, [WBD+94], p113, but it is not clear in practice that this is indeed the behaviour of the Aldor compiler. This is discussed in further detail in Sections 7 and 14.

Aldor allows *overloading*: The same name can be defined more than once, provided the types resolve any ambiguity. For example, the definitions below give two meanings to `x`, one to `x` as an `Integer` and one to `x` as a `Boolean`:

```
x : Integer == 5;
x : Boolean == true;
```

An important consequence of overloading is that terms can have more that one type. For instance, in the context above `x` has both type `Integer` and type `Boolean`. As a consequence $\vdash$ is not a function from contexts and terms to types, but is really a relation.

Finally, contexts can contain *import's*, e.g. `import from Integer`, which cause a whole set of names in a domain (or library) to be imported into the current scope. The rules concerning such import-statements will be given in subsection 7 when we consider domains. Other aspects of the import mechanism are discussed in Sections 7.1.2 and 16.

[2]N.B. Recall that we disregard all the imperative features of Aldor, and only consider the functional part of Aldor. So when we talk about variables these are never variables in the sense of imperative programming – i.e. memory locations – but always variables in the mathematical sense of 'fixed but arbitrary values'.

Raw Syntax

The grammar below defines the raw syntax of Aldor-contexts:

$$
\begin{array}{lcll}
\Gamma \in \mathit{Context} & ::= & \epsilon & \text{the empty context} \\
 & & \mid \Gamma; x : T & \text{declaration} \\
 & & \mid \Gamma; x : T\texttt{==}t & \text{“typed” definition} \\
 & & \mid \Gamma; x\texttt{==}t & \text{“untyped” definition}
\end{array}
$$

Contexts can also contain `import` statements, but these involve domains and will be treated in Section 7.

Typing Rules: well-formedness of contexts

Contexts will have to be well-formed – written $\Gamma \vdash \mathit{ok}$ –, meaning that all the terms occurring in them are well-formed, and that each constant definition has the type that is declared for it, and that any overloading in the context does not introduce ambiguities.

Rules for the well-formedness of contexts are:

$$\frac{}{\epsilon \vdash \mathit{ok}}\ \epsilon \text{ ok}$$

$$\frac{\Gamma \vdash \mathit{ok} \qquad \Gamma \vdash T : \texttt{Type} \qquad (x:T) \notin \Gamma}{\Gamma; x : T \vdash \mathit{ok}}\ \text{declaration ok}$$

$$\frac{\Gamma \vdash \mathit{ok} \qquad \Gamma \vdash t : T \qquad (x:T) \notin \Gamma}{\Gamma; x : T\texttt{==}t \vdash \mathit{ok}}\ \text{typed definition ok}$$

$$\frac{\Gamma \vdash \mathit{ok} \qquad \Gamma \vdash t : T \qquad \Gamma \mathrel{!\vdash} t : T \qquad (x:T) \notin \Gamma}{\Gamma; x\texttt{==}t \vdash \mathit{ok}}\ \text{untyped definition ok}$$

Some points to note here

- The premiss $(x : T) \notin \Gamma$ is shorthand for saying that Γ does not already contain a declaration or definition for `x` of type `T`. Multiple declarations or definitions of x are only allowed if the types of these x's are all different. (Note that here the notion of *equality of types* plays a role. More on that in 14.)
- The notation $\Gamma \mathrel{!\vdash} t : T$ is used as shorthand for 'T is the only type derivable for the term t in the context Γ (up to type equality)'. This is a premiss of a definition without an explicit type.

  Such a definition, which has the form $x\texttt{==}t$, is only allowed if there is only one possible type for t in the particular context Γ. If t has more than one type in Γ due to overloading then one of these types has to be explicitly given in the definition, which will then be of the form $x : T\texttt{==}t$.
- It would be nice to consider only definitions of the form $x : T\texttt{==}t$ in the formalisation here, and just treat definitions of the form $x\texttt{==}t$ as shorthand or syntactic sugar. However, it turns out that there are differences between the two forms of definitions with regards to equality, (which will be discussed in Section 14).

We have the obvious rules for using declarations and definitions in the context:

$$\frac{\Gamma; x : T; \Gamma' \vdash ok}{\Gamma; x : T; \Gamma' \vdash x : T} \text{ use declaration}$$

$$\frac{\Gamma; x : T\texttt{==}t; \Gamma' \vdash ok}{\Gamma; x : T\texttt{==}t; \Gamma' \vdash x : T} \text{ use typed definition}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma; x\texttt{==}t; \Gamma' \vdash ok}{\Gamma; x\texttt{==}t; \Gamma' \vdash x : T} \text{ use untyped definition}$$

All the typing rules we introduce in this report will require that contexts are well-formed. Because it is annoying to always have to include this premiss explicitly *from now on we implicitly assume that all contexts are well-formed.*

The scope rules of Aldor are quite complex: the whole of Chapter 8 in the manual is dedicated to them. On the other hand, the scope mechanism is largely independent of type issues; once scopes are delimited, type checking is done within those scopes. The mechanisms do interact, for example, when default arguments are present, but we do not treat that feature of Aldor in this document.

Differences with the Aldor compiler

The Aldor compiler does not behave exactly as prescribed by the rules for contexts given above:

- Sometimes the Aldor compiler is more strict than necessary, and does not accept contexts which are well-formed by the rules above. For example, the Aldor compiler rejects

  ```
  y : Integer == 5;
  y == Boolean;
  ```

- Sometimes the Aldor compiler accepts ambiguous contexts which are not well-formed by the rules above. For example, the Aldor compiler allows

  ```
  x == 5;
  x : Integer == 7;
  ```

  This should be rejected, as it clearly introduces an ambiguity. So this is really an bug in the Aldor compiler (or in the Aldor language.)

- Contrary to what one would expect, typed and untyped definitions are treated differently by the Aldor compiler. Replacing one by the other in the examples above leads to different behaviour of the compiler; in particular, the Aldor compiler accepts

  ```
  y : Integer == 5;
  y : Type == Boolean;
  ```

  and rejects

  ```
  x : Integer == 5;
  x : Integer == 7;
  ```

To summarise, the anomalies discussed here arise from untyped declarations (such as `x == t`) rather than typed ones (like `x:T == t`).

5 Types and Type Tuples

5.1 `Type`

As mentioned earlier, the types themselves also have types. Namely, there is a type of all types, written `Type`. The syntax and typing rule are simple.

Raw Syntax

$$\begin{array}{rcll} t, T \in \mathit{Term} & ::= & \ldots & \\ & & \mid \texttt{Type} & \text{the type of all types} \end{array}$$

Typing Rules

$$\boxed{\dfrac{}{\Gamma \vdash \texttt{Type} : \texttt{Type}}\ \texttt{Type}\ \text{form}}$$

5.2 Type Tuples

Type tuples can be sequences of types

$$(T_1, \ldots, T_n)\ ,$$

sequences of *declarations*

$$(x_1 : T_1, \ldots, x_n : T_n)\ ,$$

or any combination of the two, e.g.

$$(x_1 : T_1, T_2, x_3 : T_3, \ldots).$$

Type tuples serve as a common building block for several type constructions, such as function types, cross products, record types, and union types. For example, function types are of the form

$$\vec{T}\texttt{->}\ldots$$

with $\vec{T}$ a type tuple.

There are two – quite different – reasons for having declarations $x : T$ in type tuples. Firstly, they make it possible to have *dependencies*, e.g.

$$(X : \texttt{Type}, x : X)$$

Secondly, they introduce *names*, which is essential in record types, e.g.

$$\texttt{Record}(x : \texttt{Integer}, y : \texttt{Integer})$$

Names are also used for the so-called keyword argument style, where arguments to a function are named (see Section 6).

Raw Syntax

$$\begin{array}{rcll} t, T \in \mathit{Term} & ::= & \ldots & \\ & & \mid \texttt{Tuple Type} & \text{the type of all type tuples} \\ \vec{T} \in \mathit{TypleTuple} & ::= & (D_1, \ldots, D_n) & \\ D & ::= & T \mid x : T & \end{array}$$

Typing Rules

$$\frac{\Gamma \vdash T_i : \texttt{Type}}{\Gamma \vdash (T_1, \ldots, T_n) : \texttt{Tuple Type}} \text{ non-dependent type tuple intro}$$

$$\frac{\Gamma; x_1 : T_1; \ldots; x_{j-1} : T_{j-1} \vdash T_j : \texttt{Type}}{\Gamma \vdash (x_1 : T_1, \ldots, x_n : T_n) : \texttt{Tuple Type}} \text{ dependent type tuple intro}$$

Some things to note here

- In our formalisation, type tuples with names for only some of the fields like $(T_1, x_2 : T_2)$ are treated as syntactic sugar, by inserting dummy names.
- N.B. Type tuples are not types, i.e. they cannot have inhabitants, and not occur to the right-hand side of ":" in a typing judgement. However, there are courtesy conversions from type tuples to cross products – which *are* types – and back, as described in Section 15. This effectively makes type tuples into types.
- The Aldor compiler does not appear to insist that the x_i are distinct in a type tuple, but it seems safer to insist that they are.
- Aldor allows even more complicated expressions as type tuples than those described here. Type tuples can also contain definitions of the form $x : T$==t. These definitions are used for default arguments of functions and default values of fields in records.

  We will not try to formalise this sort of definition, since default values can be dealt with as 'syntactic sugar' which is removed prior to type analysis.
- Aldor in fact treats `Tuple Type` as an instance of the general `Tuple` construction, which will be discussed in section 9. However, doing this causes serious complications – discussed below – so we prefer to describe `Tuple Type` here separately.
- The main question about type tuples is in how far they are treated as first-class citizens. Do type tuples only occur as subexpressions of larger expressions, or can they also occur as expressions on their own, passed around as parameters, etc.? And a related question is whether `Tuple Type` is a first-class type, i.e. whether `Tuple Type:Type`.

  The Aldor compiler, in keeping with the spirit of the types-as-values approach, treats `Tuple Type` as an ordinary type and (hence) type tuples as first-class citizens. Our formalisation does not. Below we discuss our reasons for not doing this.

  Type tuples are a useful building block for several type constructions. For example, an n-ary function type is of type $(T_1, \ldots, T_n) \rightarrow \ldots$, and an n-ary cross product is of type $\texttt{Cross}(T_1, \ldots, T_n)$. Treating type tuples as first-class citizens makes it possible to give very compact descriptions for these constructions. For instance, in Aldor the type constructor `Cross` can be typed as follows

  ```
  Cross : Tuple Type -> Type
  ```

However, treating type tuples as first-class citizens in this way has serious disadvantages.

Having type tuples as first-class citizens and having `Tuple Type` as a first-class type, would mean that type tuples can be passed around as arguments, and that we can have variables X : `Tuple Type`. But then there can be records r : `Record` X for which we do not statically know their fields, and functions $f : X$`->Integer` for which we do not statically know their arity.

On the other hand, the only kind of functions we can write over types such as these will be unable to analyse the type tuples at all, so they will resemble the 'parametric polymorphic' functions of languages like SML and Haskell. We therefore do not deal with this aspect of type tuples in this treatment.

6 Functions

There are several kinds of functions in Aldor:

- simple unary functions, e.g. `f :  Integer ->Integer`.
- n-ary functions, e.g. `binaryf :  (Integer,Integer) ->Integer`
- dependent functions, e.g. `fdep :  (R:Ring) ->(R ->R)`.
- n-ary dependent functions, e.g. `f2dep :  (R:Ring,x:R) ->R`. [3]
- functions can have default arguments, e.g. `fdefault:(n:Integer==0) ->Integer`.

There are also functions which return so-called "multiple values", but as mentioned before we do not consider multiple values in our formalisation.

Expressions are formed in a number of ways, most of which are variants of function or operator application. The typing rules for function application are therefore central to explaining the typing of computations in the functional (or equivalently *applicative*) subset of Aldor.

There are several ways of passing arguments to functions in applications:

- normal arguments, e.g. `f(5)` or `binaryf(3,8)`,
- arguments by keyword, e.g. `f2dep(R==Integer,x==0)`,
- default arguments, e.g. `fdefault()`.

We will not consider default arguments in the formal description, but we will consider keyword arguments. These may seem a bit baroque to include in the formalisation, but other type constructions, notably records and unions, crucially depend on this.

6.1 Simple function types S->T

First we consider the simplest form of functions, namely unary functions with types of the form T_1->T_n.

[3] Note that here not only the type of the output depends on an input, but also the type of the second input depends on first input.

Raw Syntax

$$\begin{array}{rcll} t, T \in \mathit{Term} & ::= & \ldots & \\ & & \mid (x : T_1) : T_2\texttt{+->}t & \text{abstraction} \\ & & \mid t_1(t_2) & \text{application} \\ & & \mid T_1\texttt{->}T_2 & \text{function type} \end{array}$$

Typing Rules

$$\frac{\Gamma \vdash S, T : \texttt{Type}}{\Gamma \vdash S\texttt{->}T : \texttt{Type}} \text{ function type formation}$$

$$\frac{\Gamma \setminus \{x\}, x : S \vdash t : T}{\Gamma \vdash ((x : S) : T\texttt{+->}t) : S\texttt{->}T} \text{ function intro}$$

$$\frac{\Gamma \vdash f : S\texttt{->}T \qquad \Gamma \vdash s : S}{\Gamma \vdash f(s) : T} \text{ function elim}$$

Some examples

```
double : Integer -> Integer
     == (n:Integer) : Integer +-> n+n ;

eight : Integer
      == double 4;
```

Discussion

- The local (bound) variable $x : S$ in a lambda abstraction hides any other occurrences of x in the context. Hence the $\Gamma \setminus \{x\}$ in the introduction rule above.
- Functions are first-class citizens, so higher-order functions – functions that have other functions as input or output – can be formed.
- The usual notation for functions in definitions is `f(x:S) :T == t`, which we treat as syntactic sugar for

  ```
  f : S->T == (x:S):T +-> t
  ```

- In Aldor function definitions can be recursive, but in our formalisation not. Allowing this would not be difficult, for this we would have to include `f:S->T` itself in the context when type-checking the body of `f`.
- In defining a (recursive) function, `fac` say, the identifier being defined can be used in an overloaded fashion, as in the example

  ```
  fac (b:Boolean) : Boolean == ~b;

  fac (n:Integer) : Integer == if fac(fac(n=0))
                                  then 1
                                  else (n*(fac (n-1)));
  ```

  where `fac` is used over both booleans and integers in the recursive definition of `fac` over `Integer`.

6.2 Dependent function types $(x : S)$->T

Now we consider unary functions with types of the form $(x : S)$->T. Such functions can be *dependent* types, where T depends on x.

Raw Syntax

$$\begin{array}{lcll} t, T \in Term & ::= & \ldots & \\ & & \mid (x : T_1)\texttt{->}T_2 & \text{dependent function type} \end{array}$$

Typing Rules

$$\frac{\Gamma \vdash S : \texttt{Type} \qquad \Gamma \setminus \{x\}, x : S \vdash T : \texttt{Type}}{\Gamma \vdash (x : S)\texttt{->}T : \texttt{Type}} \ \text{dependent function formation}$$

$$\frac{\Gamma \setminus \{x\}, x : S \vdash t : T}{\Gamma \vdash ((x : S) : T\texttt{+->}t) : (x : S)\texttt{->}T} \ \text{dependent function intro}$$

$$\frac{\Gamma \vdash f : (x : S)\texttt{->}T \qquad \Gamma \vdash s : S}{\Gamma \vdash f(s) : T[x := s]} \ \text{dependent function elim}$$

The notation $T[x := s]$ is used for the result of substituting s for every free occurrence of x in T.

Some examples

```
polyId :  (T:Type) -> (T -> T)
       == (T:Type) : T->T
                 +-> (x:T) : T +-> x;

idIntArrowInt  :  (Integer -> Integer) -> (Integer -> Integer)
               == polyId (Integer->Integer);
```

Discussion

- Note that now there can be bound variables in types! Substitution in types is needed: $T[x := s]$ denotes T with all free occurrences of x replaced by s.
- We can now build parametric polymorphic functions as in system F, and similar to those in modern functional programming languages like ML or Haskell, but with explicit polymorphism, i.e. polymorphic functions get explicit type parameters.
- The core of the Aldor type system, containing only $\texttt{Type} : \texttt{Type}$ and the rules for functions above can be described as the Pure Type System (PTS) [Bar93], namely the PTS with the specification

  $$S = \{\texttt{Type}\}, A = \{\texttt{Type} : \texttt{Type}\}, R = \{(\texttt{Type}, \texttt{Type})\}$$

  (except that in Aldor we do not have β-equality for types – more on that in Section 15). Note that any PTS can be mapped into the PTS above, so that this PTS is as expressive as any other PTS, including for instance the Calculus of Constructions [CH88] or the Extended Calculus of Constructions [Luo89].

- The Aldor compiler has problems with dependent types as first-class citizens, and crashes if we pass these as arguments to functions. This seems to be a bug.

6.3 N-ary functions

Aldor allows n-ary functions, i.e. functions that get more than one argument at once. Such functions can also have dependent types. Here the notion of type tuple is used: n-ary functions have types of the form $\overrightarrow{T}$->T, where $\overrightarrow{T}$ is a type tuple. This means these types are of the form $(S_1, \ldots, S_n)$->T or $(x_1 : S_1, \ldots, x_n : S_n)$->$T$. We only consider the latter form, and treat the former as a special case.

Raw Syntax

$$
\begin{array}{lcll}
t, T \in Term & ::= & \ldots & \\
& & \mid (x_1 : T_1, \ldots, x_n : T_n) : T\texttt{+->}t & \text{n-ary abstraction} \\
& & \mid t(t_1, \ldots, t_n) & \text{n-ary application} \\
& & \mid \overrightarrow{T}_1\texttt{->}T & \text{n-ary (dependent) function type}
\end{array}
$$

Typing Rules

Let $\overrightarrow{S} \equiv (x_1 : S_1, \ldots, x_n : S_n)$, and in the rules and subsequent discussion we assume that all the x_i are distinct.

$$
\frac{\Gamma \vdash \overrightarrow{S} : \texttt{Tuple Type} \qquad \Gamma \setminus \{x_1, \ldots, x_n\}, \overrightarrow{S} \vdash T : \texttt{Type}}{\Gamma \vdash \overrightarrow{S}\texttt{->}T : \texttt{Type}} \text{ n-ary function type formation}
$$

$$
\frac{\Gamma \setminus \{x_1, \ldots, x_n\}, \overrightarrow{S} \vdash t : T}{\Gamma \vdash \overrightarrow{S} : T\texttt{+->}t : \overrightarrow{S}\texttt{->}T} \text{ n-ary function intro}
$$

$$
\frac{\Gamma \vdash f : \overrightarrow{S}\texttt{->}T \qquad \Gamma \vdash s_i : S_i[x_1 := s_1, \ldots, x_{i-1} := s_{i-1}] \text{ for all } 1 \leq i \leq n}{\Gamma \vdash f(s_1, \ldots, s_n) : T[x_1 := s_1, \ldots, x_n := s_n]} \text{ n-ary function elim}
$$

Some examples

```
polyCompose (S:Type, T:Type, U:Type, f:T->U, g:S->T)
            :  S->U
            == (x:S) : U +-> (f (g x));

quadruple : Integer -> Integer
         == polyCompose (Integer,Integer,Integer,double,double) ;
```

Discussion

- The context $\Gamma \setminus \{x_1, \ldots, x_n\}, \overrightarrow{S}$ contains a type tuple $\overrightarrow{S}$. The meaning is the obvious one, namely the context $\Gamma \setminus \{x_1, \ldots, x_n\}$ extended with all the declarations in $\overrightarrow{S}$.
- If we identify the type tuple (S) with the type S, we get the rules for unary functions as a special case of these rules.
- Although in practice n-ary functions are very useful, they do not fundamentally increase the power of the type system. We could have omitted them in the formal description here, and treat them as syntactic sugar.

  In fact, an n-ary function type $(x_1 : S_1, \ldots, x_n : S_n)\texttt{->}T$ could almost be treated as syntactic sugar for $\texttt{Cross}(x_1 : S_1, \ldots, x_n : S_n)\texttt{->}T$. However, in the former type T can depend on the x_i, in the latter it cannot.
- Is the $(s_1, \ldots, s_n)$ in the elimination rule "a multiple value"? Yes, this seems to be the case, as we can pass a cross product to a function as argument. See cross products (Section 10).
- There are also hybrid forms of keyword argument and normal arguments. We ignore these.
- The Aldor compiler allows the same variable name to occur more than once in the domain of an n-ary function type. E.g. it accepts

  ```
  tt : Type == (x:Integer,x:Boolean) -> Integer
  ```

  We see no substantial need for this and so our formalisation does not allow this.
- We could introduce some syntax for auxiliary judgements of the form $\Gamma \vdash \overrightarrow{s} : \overrightarrow{S}$ to simplify the typing rules, e.g. to

  $$\frac{\Gamma \vdash f : \overrightarrow{x : S}\texttt{->}T \qquad \Gamma \vdash \overrightarrow{s} : \overrightarrow{S}}{\Gamma \vdash f\,\overrightarrow{s} : T[\overrightarrow{x} := \overrightarrow{s}]} \text{ function elim}$$

  However, to avoid possible confusion about the status of type tuples $\overrightarrow{S}$ and term tuples $\overrightarrow{s}$ here we do not do this.

6.4 Keyword arguments

For applications of functions with types of the form $(x_1 : S_1, \ldots, x_n : S_n)\texttt{->}T$ the so-called keyword argument style can be used. Such an application is of the form $f(x_{j_1}\texttt{==}s_{j_1}, \ldots, x_{j_n}\texttt{==}s_{j_n})$. Here the parameters do not have be given in any particular order, but the labels tell which is which.

Raw Syntax

$$\begin{array}{lcll} t, T \in \mathit{Term} & ::= & \ldots & \\ & = & t(x_1\texttt{==}t_1, \ldots, x_n\texttt{==}t_n) & \text{application with keyword argument} \end{array}$$

Typing Rules

$$\frac{\begin{array}{l}\Gamma \vdash f : (x_1 : S_1, \ldots, x_n : S_n)\texttt{->}T \\ \Gamma \vdash s_i : S_i[x_1 := s_1, \ldots, x_{i-1} := s_{i-1}] \text{ for all } i\end{array}}{\Gamma \vdash f(x_1\texttt{==}s_1, \ldots, x_n\texttt{==}s_n) : T[x_1 := s_1, \ldots, x_n := s_n]} \text{ keyword argument}$$

Keyword arguments play an import role later in the description of records and unions.

7 Domains

There are two kinds of domains, *packages* and *abstract datatypes (ADT's)*.

- Packages are of the form

  $$\texttt{add}\{x_1 : T_1\texttt{==}t_1; \ldots; x_n : T_n\texttt{==}t_n\}$$

  So a package consists of a collection of definitions. By default, the names defined in a package are called its *exports*; it is possible explicitly to control the exports of a package.

- ADT's are of the form

  $$\texttt{add}\{\texttt{Rep==>}T; x_1 : T_1\texttt{==}t_1; \ldots; x_n : T_n\texttt{==}t_n\}$$

  so every ADT provides a distinguished type `Rep` as export.

The types of domains – called *categories* – are of the form

$$\texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\}$$

and are expressions of type `Category`.

The types T_i in domains are optional and can be left out. This causes some complications, as components of the form `x == t` and of the form `x:T == t` are treated differently[4]. These differences start playing a role when there are dependencies between the components of a domain. For this we make a distinction between

- (simple) packages, where none of the t_i or T_i depends on an x_j,
- dependent packages, where some of the t_i or T_i do depend on other x_j's,

and a further distinction for dependent packages between

- type-dependent packages, where some $t_i : T_i$ depend on the *type* of other x_j's,
- definition-dependent packages, where some $t_i : T_i$ depend on the *definition* and on the *type* of other x_j's.

Below we start with the simplest form of package and then introduce further complexities in stages.

[4]namely, they are treated differently with regard to equality; see Section 14.

7.1 Packages

7.1.1 Simple packages, no dependencies

The simplest form of domain is a package. This is essentially just a record.

Raw Syntax

$$
\begin{array}{lcll}
t, T \in \mathit{Term} & ::= & \ldots & \\
 & & \mid \texttt{add}\{x_1 : T_1\texttt{==}t_1; \ldots; x_n : T_n\texttt{==}t_n\} & \text{package} \\
 & & \mid x_i\$x & \text{projection from package} \\
 & & \mid \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\} & \text{category (type of a package)}
\end{array}
$$

Typing Rules

$$\frac{\Gamma \vdash T_i : \texttt{Type} \quad \text{all pairs } x_i : T_i \text{ distinct in } \Gamma}{\Gamma \vdash \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\} : \texttt{Category}} \text{ category form}$$

$$\frac{\Gamma \vdash t_i : T_i \quad \text{all pairs } x_i : T_i \text{ distinct in } \Gamma}{\Gamma \vdash \texttt{add}\{x_1 : T_1\texttt{==}t_n; \ldots; x_n : T_n\texttt{==}t_n\} : \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\}} \text{ package intro}$$

$$\frac{\Gamma \vdash x : \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\}}{\Gamma \vdash x_i\$x : T_i} \text{ package elim}$$

Some examples

```
c : Category
  == with {zero:Integer; one:Integer};

p : with {zero:Integer; one:Integer}
  == add {zero:Integer==0, one:Integer==1};

project : Integer == zero$p + one$p ;
```

Discussion

- Variables can be overloaded in a package, provided – as usual – they have distinct types. Hence the premiss "all pairs $x_i : T_i$ distinct in Γ" above.

  This restriction does not seem to apply to category expressions; for example the Aldor compiler accepts

  ```
  ttt : Category ==  with {zero : Integer; zero : Integer}
  ```

  However, it seems better not to allow this.

- In the elimination rule, the package we project from has be a variable – i.e. a package name– it cannot be an `add`-expression. (Note that this is already enforced by the grammar for terms.)

- Aldor accepts domains and categories written with "," instead of ";", e.g. of the form $\texttt{add}\{x_1 : T_1\texttt{==}t_1, \ldots, x_n : T_n\texttt{==}t_n\}$. However, the typing behaves weirdly, and it is not clear what the intended meaning of such domains and categories might be.

- Note that packages are essentially records. (However, when we take the imperative features of Aldor into account then there are differences between packages and records. For records the fields can be updated imperatively, for packages not.)
- There is subtyping on categories, which will be discussed in Section 15.
- There is more syntax for domains and categories, which we ignore. For instance, there are domain-extensions of the form

  $$d\ \texttt{add}\{x_1 : T_1\texttt{==}t_1; \ldots; x_n : T_n\texttt{==}t_n\}$$

  where d is the name of a domain. According to the Aldor User Guide such domains can be regarded as shorthand for the domain that includes both the definitions contained in d and the x_i.

  Similarly, we ignore *category extensions* of the form

  $$c\ \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\}$$

  and *category joins* of the form

  $$\texttt{join}(c_1, \ldots, c_n)$$

  which can also be regarded as syntactic sugar.

7.1.2 The `import` statement

The `import`-statement provides an alternative to the explicit projections of the form $x_i\$p$. By `import`-ing a domain `p` into the context, we can refer to its components as x_i instead of $x_i\$p$ (provided this does not introduce ambiguities).

Raw Syntax

$$\begin{array}{lcll} \Gamma \in \mathit{Context} & ::= & \ldots & \\ & & \mid \Gamma; \texttt{import from } d & \text{domain import} \end{array}$$

Typing Rules

$$\frac{\Gamma \vdash d : \texttt{with}\{\ldots\}}{\Gamma; \texttt{import from } d \vdash \mathit{ok}}$$

$$\frac{\begin{array}{c}\Gamma \vdash d : \texttt{with}\{\ldots x_i : T_i \ldots\} \quad (x_i : T_i) \notin \Gamma \\ x_i \text{ not } \texttt{import}\text{-ed from another package in } \Gamma; \Gamma'\end{array}}{\Gamma; \texttt{import from } d; \Gamma' \vdash x_i : T_i}\ \text{import}$$

Example

```
p : with {zero:Integer; one:Integer}
  == add {zero:Integer==0, one:Integer==1};

import from p;

project' : Integer == zero + one ;
```

- If the contexts imports two or more packages that have `x` as an export, then the resulting `x` is overloaded and any use of the symbol `x` will be disambiguated by type. If any two of the definitions have the same type then we have to use explicit projections of the form `x$p` to tell which one we mean.

7.1.3 Simple dependent packages

The rule for package introduction given earlier does not allow for dependencies between the different components. The slightly more complicated introduction rule below allows the t_i to refer to earlier x_j:

$$\frac{\Gamma; x_1 : T_1; \ldots ; x_{i-1} : T_{i-1} \vdash t_i : T_i}{\Gamma \vdash \mathtt{add}\{x_1 : T_1 \mathtt{==} t_1; \ldots ; x_n : T_n \mathtt{==} t_n\} : \mathtt{with}\{x_1 : T_1; \ldots ; x_n : T_n\}} \text{ package intro}$$

Using this rule it is possible to make packages where some components are defined in terms of other components. For example,

```
dp : with{x:Integer; y:Integer}
   == add {x:Integer == 5; y:Integer == x};
```

Note that we only have a weak form of dependency here:

- To check that `y:Integer == x` is well-typed, only the type of `x` – i.e. `Integer` has to be visible. The definition of `x` – i.e. `5` – is not needed.
- Also, no dependency shows up between the types of the fields of the package, i.e. in $\mathtt{with}\{\mathtt{x} : \mathtt{Integer}; \mathtt{y} : \mathtt{Integer}\}$.
- The typing rule does not allow mutual dependencies, but Aldor actually does allow this.

This form of (weak) dependency is found in a number of existing languages, such as SML; what are usually termed 'dependent types' are discussed in the next subsection.

7.1.4 Dependent Packages and Dependent Categories

An example of a truly dependent package is

```
add{X:Type == Integer; x:X == 5}
```

Note that here we have a stronger form of dependency than in the earlier example above:

- To check that `x:X == 5` is well-typed the definition of `X` – `Integer` – is needed. Just knowing the type of `X` – `Type` – is not enough.
- The dependency shows up between the types of the fields of the package, which would be $\mathtt{with}\{\mathtt{X} : \mathtt{Type}; \mathtt{x} : \mathtt{X}\}$.

However, there are three "anomalies" with such dependent packages in Aldor:

- The Aldor compiler does not accept the dependent package above. We have to write it as follows instead

```
add{X == Integer; x:X == 5}
```

So the two ways of writing fields – `X == t` and `X : T == t` – are not equivalent; we have to use the former for the definition of `X` to be "visible". (The presence or not of the modifier `define` appears to have no effect in this context.)

The dependent package works fine now. For example:

```
d == add{X == Integer; x:X == 5};

projX : Type == X$d;
projx : X$d == x$d

import from d;
projX2 : Type == X;
projx2 : X == x;
```

- There is a further problem when it comes to typing dependent packages: The Aldor compiler does not accept

  $$\mathtt{with}\{\mathtt{X} : \mathtt{Type}; \mathtt{x} : \mathtt{X}\}$$

  as the type of

  $$\mathtt{add}\{\mathtt{X} == \mathtt{Integer}; \mathtt{x} : \mathtt{X} == \mathtt{5}\}$$

  (even though it does accept `with{ X:Type; x:X }` as well-formed category). To type the dependent package above, the definition of `X` in its type has to be expanded

  ```
  add{X == Integer; x:X == 5} : with{x:Integer == 5}
  ```

  But now the `X`-field of the domain will not be visible as an export.

  The types for `add{X == Integer; x:X == 5}` discussed so far represent two extremes of generality; it might be supposed that there is an intermediate candidate, but none of the following types is accepted by the Aldor compiler as a valid type for `add{X == Integer; x:X == 5}`:

  ```
  with{X:Type == Integer; x:X}
  with{X:Type == Integer; x:Integer}
  with{X == Integer; x:X}
  with{X == Integer; x:Integer}
  ```

- The typing of dependent packages as discussed in the previous point has some undesirable consequences. As soon as we give an explicit type to a dependent package, some of the fields (namely the ones that other fields depend on) are no longer visible as exports.

  For example, if we define

  ```
  d == add{X == Integer; x:X == 5}
  ```

  then we can access both `d$X` and `d$x`, but if we define

  ```
  d' == add{X == Integer; x:X == 5} : with{x:Integer == 5}
  ```

then we can not access the `X`-field of the domain `d'`. So in the definition of `d'` above `X == Integer` is essentially just a (local) macro.

The problem with the invisibility of certain fields only occurs as soon an explicit type is given to a dependent domain. This happens in the definition of `d'` above, but not in the definition of `d`. However, it also happens as soon a dependent package such as `d` is passed as a argument to a function: such a function will have to declare a parameter of type $\mathtt{with}\{\mathtt{x} : \mathtt{Integer}\}$ (we cannot pass `d` to a function expecting a parameter of type $\mathtt{with}\{\mathtt{X} : \mathtt{Type}; \mathtt{x} : \mathtt{X}\}$)) and in the function we do not have access to any `X`-field.

All this means that *dependent domains are not really usable as first-class citizens*; we could reinterpret this to say that Aldor does not have first-class modules. Still, if we want to treat libraries as dependent packages this is not a problem.

Raw Syntax

$$
\begin{array}{rcll}
t, T \in Term & ::= & \ldots & \\
& & \mid \mathtt{add}\{D_1; \ldots; D_n\} & \text{dependent package} \\
D & ::= & x\mathtt{==}t \mid x : T\mathtt{==}t &
\end{array}
$$

Typing Rules

$$
\dfrac{\begin{array}{ll}
\text{for all } D_i \text{ of the form } x_i\mathtt{==}t_i : & \Gamma; D_1, \ldots, D_{i-1} \vdash t_i : T \text{ for some } T \\
\text{for all } D_i \text{ of the form } x_i : T_i\mathtt{==}t_i : & \Gamma; D_1, \ldots, D_{i-1} \vdash t_i : T_i \\
\text{All } x_i : T_i \text{ distinct in the context } \Gamma &
\end{array}}{\Gamma \vdash \mathtt{add}\{D_1, \ldots, D_n\} : \mathtt{with}\{x_i : T_i^* \mid D_i \equiv x_i : T_i\mathtt{==}t_i\}} \quad \text{dependent package intro}
$$

where T_i^* is short for $T_i[x_{i-1} := t_{i-1}] \ldots [x_1 := t_1]$

$$
\dfrac{\Gamma \vdash x : \mathtt{with}\{\ldots; x_i : T_i\mathtt{==}t_i; \ldots\}}{\Gamma \vdash x_i\$x : T_i} \quad \text{dependent package elim}
$$

$$
\dfrac{\begin{array}{l}
\Gamma \vdash d : \mathtt{with}\{\ldots x_i : T_i \ldots\} \quad (x : T_i) \notin \Gamma \\
x_i \text{ not } \mathtt{import}\text{-ed from another package in } \Gamma; \Gamma'
\end{array}}{\Gamma; \mathtt{import\ from\ } d; \Gamma' \vdash x_i : T_i} \quad \text{dependent import}
$$

The requirement in the first rule that all $x_i : T_i$ distinct in the context Γ is to ensure unique typing of each x_i in the context Γ. Formally we have to ensure that $T_i^* \neq T_j^*$ for any pair $x_i : T_i(\mathtt{==}t_i)$ and $x_j : T_j(\mathtt{==}t_j)$ where x_i and x_j are the same name.

Discussion

- The elimination and import rule above are identical to those given earlier, for non-dependent packages. Here the fact that any dependencies get "expanded away" in the introduction rule is an advantage. If one were to allow the typing

```
d : with{X:Type; x:X} == add{X == Integer; x:X == 5}
```

then projecting the `x`-field of `d` would require a substitution (as in the elimination rules given below) as `x$d:X$d` and not `x$d:X`[5].

- The Aldor compiler has problems with dependent packages and crashes when these become complicated. This seems to be a bug.

As long as we don't give an explicit type to a dependent domain its x_i==t_i fields as well as its $x_i : T_i$==t_i fields are accessible:

$$\frac{\Gamma; D \vdash t_i : T_i}{\Gamma; x\texttt{==add}\{D; x_i\texttt{==}t_i; D'\}, \Gamma' \vdash x_i\$x : T_i^*} \text{ dependent package elim1}$$

$$\frac{\Gamma; D \vdash t_i : T_i}{\Gamma; x\texttt{==add}\{D; x_i : T_i\texttt{==}t_i; D'\}, \Gamma' \vdash x_i\$x : T_i^*} \text{ dependent package elim2}$$

where T_i^* is short for $T_i[x_{i-1} := x_{i-1}\$x] \ldots [x_1 := x_1\$x]$.

Some examples

First, a dependent package without explicit type

```
dependentPackage == add { X == Integer ;
                          x : X == 0 ;
                          f : X -> X == (n:X):X +-> (n+1) };

projX  : Type               == X$dependentPackage ;
projx  : X$dependentPackage == x$dependentPackage ;
projx2 : Integer            == x$dependentPackage ;

projf1 : X$dependentPackage -> X$dependentPackage == f$dependentPackage ;
projf2 :            Integer -> Integer            == f$dependentPackage ;

import from dependentPackage;

importX  : Type    == X;
importx  : X       == x ;
importx2 : Integer == x ;

importf1 :       X -> X       == f$dependentPackage ;
importf2 : Integer -> Integer == f$dependentPackage ;
```

Note that `X$dependentPackage` or `X` and `Integer` are really treated as equal.

Now, a dependent package for which we give an explicit type

```
typedDependentPackage : with{z:Integer}
                     ==  add{Z == Integer ;
                             z : Z == 0};

projz  : Integer == z$typedDependentPackage;

import from typedDependentPackage;

importz : Integer == z
```

[5] Similarly, the notion of subtyping would become more complicated.

The Aldor compiler rejects any use of `Z$typedDependentPackage` or `Z` here. So the definition `Z == Integer` is effectively nothing but a macro local to the body of the package, which get expanded away as soon as we leave this scope.

7.1.5 Dependent Categories

Aldor allows the formation of dependent categories :

$$\frac{\Gamma; x_1 : T_1; \ldots ; x_{i-1} : T_{i-1} \vdash T_i : \texttt{Type}}{\Gamma \vdash \texttt{with}\{x_1 : T_1; \ldots ; x_n : T_n\} : \texttt{Category}} \text{ dependent category form}$$

For instance, this rule allows the formation of

```
with{x:Type; y:x}
```

But Aldor does not allow such dependent categories to be used as types of the dependent domains discussed in the previous section! So, it seems that there is little point in allowing dependent categories.

7.2 Abstract Data Types

Abstract data types are like packages, but they contain a definition of a type `Rep`, which gives the representation type for the abstract type introduced by the ADT.

Raw Syntax

$$\begin{array}{lcll}
t, T \in Term & ::= & \ldots & \\
& & \mid \texttt{add}\{\texttt{Rep==>}T; x_1 : T_1\texttt{==}t_1; \ldots ; x_n : T_n\texttt{==}t_n\} & \text{an ADT} \\
& & \mid x\$d & \text{projection from ADT} \\
& & \mid \% \mid \texttt{Rep} & \text{special type variables} \\
& & \mid \texttt{rep} \mid \texttt{per} & \text{special term variables} \\
& & \mid \texttt{with}\{x_1 : T_1; \ldots ; x_n : T_n\} & \text{category (type of ADT)}
\end{array}$$

The special type variable `%` is used to refer to the abstract type introduced by an ADT, and `Rep` is used to refer to the concrete representation. The special term variables `per` and `rep` are used to refer to the functions that map concrete values to abstract values and vice versa.

Note that the only way of telling that a category $\texttt{with}\{x_1 : T_1; \ldots ; x_n : T_n\}$ is the type of an ADT as opposed to the type of a package is that the special type variable `%` is used in the T_i. [6]

[6] This is unfortunate – for example in the elimination rules below it is not explicit that these apply to abstract data types and not to packages; it is therefore possible that it introduces inaccuracies into our formalisation. It might well be better to introduce some syntactic distinction between them, writing $\texttt{with}_{ADT}\{\ldots\}$ for an implementation of an abstract data type.

Typing Rules

In the rules which follow X is used to range over *names* of abstract data types and not arbitrary expressions denoting ADTs.

$$\frac{\Gamma; \% : \texttt{Type} \vdash T_i : \texttt{Type}}{\Gamma \vdash \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\} : \texttt{Category}} \text{ ADT-category-form}$$

$$\frac{\Gamma \vdash T : \texttt{Type} \qquad \Gamma; \texttt{Rep==}T, \% : \texttt{Type}, \texttt{per} : T\texttt{->}\%, \texttt{rep} : \%\texttt{->}T \vdash t_i : T_i}{\Gamma \vdash \texttt{add}\{\texttt{Rep==>}T; x_i : T_i\texttt{==}t_i\} : \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\}} \text{ ADT intro}$$

$$\frac{\Gamma \vdash X : \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\}}{\Gamma; \texttt{import from } X; \Gamma' \vdash X : \texttt{Type}} \text{ ADT elim1 - ADT is a type}$$

$$\frac{\Gamma \vdash X : \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\}}{\Gamma; \texttt{import from } X; \Gamma' \vdash x_i\$X : T_i[\% := X]} \text{ ADT elim2}$$

$$\frac{\Gamma \vdash X : \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\} \quad x_i \text{ not } \texttt{import}\text{-ed from another package in } \Gamma; \Gamma'}{\Gamma; \texttt{import from } X; \Gamma' \vdash x_i : T_i[\% := X]} \text{ ADT elim3}$$

Some examples

```
adtType : Category
        == with { x : % } ;

    adt : adtType
        == add { Rep ==> Integer ; x : % == per 0 } ;

projx : adt == dep$adt

import from adt;
importx : adt == x
```

Discussion

- The stipulation that X has to be a name refers back to the discussion in Section 3.
- Note that all the elimination rules insist that an adt X is explicitly imported, even the one for explicit projection of the from $x_i\$X$; This is done because X is needed in the type, i.e. in $T_i[\% := X]$.
- There are three names for types that play a role inside an adt (and not two, as you'd expect), namely
  - `%`, the abstract type
  - the concrete type, or representation type, e.g. `Integer`
  - `Rep`, another name for the concrete type

  `Rep==>`T is both a (local) macro , defining `Rep` as abbreviation for T, and declares the T as the concrete representation type.

- We can have untyped instead of typed definitions in ADT's. This does not make any difference.
- We could consider more complicated form of ADT's, with more dependencies than just on %.

8 Categories and category tuples

8.1 `Category`

`Category` is the type of all categories, just like `Type` is the type of all types.

Raw Syntax

$$\begin{array}{lll} t, T \in Term & ::= & \ldots \\ & & \mid \texttt{Category} \quad \text{the type of all categories} \end{array}$$

Typing Rules

$$\boxed{\frac{}{\Gamma \vdash \texttt{Category} : \texttt{Type}} \; \texttt{Category}\text{ form}}$$

`Category` is in fact a subtype of `Type`– more on that in Section 15.

8.2 Category Tuples

Just like we can make type tuples of the form $(x : T_1, \ldots, x_n : T_n)$ we can make category tuples of the form $(x : T_1, \ldots, x_n : T_n)$. We do not include these in the formalisation however, as it is not clear if these can be used for anything[7].

9 Tuples

Tuples in Aldor are homogeneous products of arbitrary length. (So one can think of them as lists.) For example

```
tt      : Tuple Integer  == (1,2,3) ;

tt1     : Tuple Integer  == (1,2,3,4,5) ;

tt2     : Integer == element(tt,2) ;
```

Raw Syntax

$$\begin{array}{llll} t, T \in Term & ::= & \ldots & \\ & & \mid (t_1, \ldots, t_n) & \text{n-tuple} \\ & & \mid \texttt{Tuple}\ T & \text{tuple type} \\ & & \mid \texttt{length} & \\ & & \mid \texttt{element} & \end{array}$$

[7] One place where they are used is for the "joins" of categories mentioned earlier, but these are excluded from the formal description.

Typing Rules

$$\frac{\Gamma \vdash T : \texttt{Type}}{\Gamma \vdash \texttt{Tuple}\ T : \texttt{Type}}\ \text{tuple formation}$$

$$\frac{\Gamma \vdash t_i : T}{\Gamma \vdash (t_1, \ldots, t_n) : \texttt{Tuple}\ T}\ \text{tuple intro}$$

$$\frac{\Gamma \vdash \texttt{Tuple}\ T : \texttt{Type}}{\Gamma \vdash \texttt{length} : \texttt{Tuple}\ T\texttt{->SingleInteger}}\ \text{tuple elim1}$$

$$\frac{\Gamma \vdash \texttt{Tuple}\ T : \texttt{Type}}{\Gamma \vdash \texttt{element} : (\texttt{Tuple}\ T, \texttt{SingleInteger})\texttt{->}T}\ \text{tuple elim2}$$

- Note that `length` and `element` above are heavily overloaded functions. Because these functions exists for all possible tuple types, they are very similar to implicitly parametric polymorphic functions as in Haskell or ML.
- Aldor considers `Tuple Type` as just another instance of this general `Tuple`-construction.

  As far as simple type tuples of the form $(T_1, \ldots, T_n)$: `Tuple Type` are concerned this is not a problem: the introduction rule for these simple type tuples is just an instance of the general introduction rule above.

  However, for type tuples of the form $(x_1 : T_1, \ldots, x_n : T_n)$ this is dubious. Aldor allows $(x : T)$: `Type` so that type tuples $(x_1 : T_1, \ldots, x_n : T_n)$ can still be regarded as instances as tuples of the form $(t_1, \ldots, t_n)$. However, the rule for dependent type tuples has to be more complicated than the one above to allow for dependencies. Also, there are problems with allowing type tuples as first-class citizens, as already discussed in Section 5.

10 Cross Products

Cross products in Aldor are heterogeneous products of a fixed arity. For example

```
pp : Cross (Integer,Boolean) == (4,true);
```

Raw Syntax

$$\begin{array}{lcll} t, T \in \mathit{Term} & ::= & \ldots & \\ & & \mid (t_1, \ldots, t_n) & \text{cross product} \\ & & \mid \texttt{Cross}(T_1, \ldots, T_n) & \text{cross product type} \end{array}$$

There seems to be no way to refer to the components of a cross product. There is no syntax such $t.i$ for the i-th component of a cross product t. This causes some problems and appears to add to the case for rationalising the various different sorts of 'products' and 'tuples' that the language contains.

Typing Rules

$$\frac{\Gamma \vdash (D_1, \ldots, D_n) : \texttt{Tuple Type}}{\Gamma \vdash \texttt{Cross}(D_1, \ldots, D_n) : \texttt{Type}} \text{ cross form}$$

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash (t_1, \ldots, t_n) : \texttt{Cross}(T_1, \ldots, T_n)} \text{ non-dependent cross intro}$$

$$\frac{\Gamma \vdash t_j : T_j[x_1 := t_1, \ldots, x_{j-1} := t_{j-1}]}{\Gamma \vdash (t_1, \ldots, t_n) : \texttt{Cross}(x_1 : T_1, \ldots, x_n : T_n)} \text{ dependent cross intro}$$

There are not really any elimination rules for cross products. The two ways to get at at the individual components of a cross product are described below[8].

1. An n-ary cross product of type $\texttt{Cross}(S_1, \ldots, S_n)$ as an argument to an n-ary function of type $(S_1, \ldots, S_n)\texttt{->}\ldots$

   $$\frac{\Gamma \vdash f : (S_1, \ldots, S_n)\texttt{->}T \qquad \Gamma \vdash s : \texttt{Cross}(S_1, \ldots, S_n)}{\Gamma \vdash f(s) : T} \text{ n-ary function elim}$$

   Note that the function f here is not a dependent function! Because we cannot refer to the components of the cross product s it is not clear how a typing rule could be given for a dependently typed function f. If s is of the form $(s_1, \ldots, s_n)$ this is not a problem, but if s is a variable, the result of a function application, etc, it is.

2. An n-ary cross product t can be taken apart into its components by a 'multiple definition' of the form

   $$(x_1, \ldots, x_n)\texttt{==}t$$

   This means the syntax for context has to be extended:

   $$\begin{array}{lll} \Gamma \in \mathit{Context} & ::= & \ldots \\ & & \mid \Gamma; (x_1, \ldots, x_n)\texttt{==}mv \quad \text{multiple value definition} \end{array}$$

   The rules for these definitions are given below.

   $$\frac{\Gamma \vdash ok \qquad \Gamma \vdash t : \texttt{Cross}(T_1, \ldots, T_n) \qquad (x_i : T_i) \notin \Gamma}{\Gamma; (x_1, \ldots, x_n)\texttt{==}t \vdash ok} \text{ typed definition ok}$$

   $$\frac{\Gamma \vdash t : \texttt{Cross}(T_1, \ldots, T_n)}{\Gamma; (x_1, \ldots, x_n)\texttt{==}t; \Gamma' \vdash x_i : T_i}$$

[8] Both actually rely on the courtesy conversion of a cross product to a multiple value. But, as we have excluded multiple values for our description of Aldor, we ignore this.

Discussion

- The Aldor compiler allows dependent cross product types, e.g.

  ```
  X : Type == Cross(X:Type,x:X)
  ```

  but it is not clear if/how we can write dependent cross products, i.e. inhabitants of such a type. For this reason we have excluded dependent cross product types from the formal description.

- The Aldor compiler in fact treats `Cross` as a function of type

  ```
  Cross : Tuple Type -> Type
  ```

  In the formalisation we choose not to do so, for two reasons. First, there are problems with treating type tuples as first-class citizens, already discussed in Section 5.2. Second, our formalisation only allows cross product types of the form $\mathtt{Cross}(T_1, \ldots, T_n)$, and excludes dependent cross product types of the form $\mathtt{Cross}(x_1 : T_1, \ldots, x_n : T_n)$, as mentioned above.

11 Records

Records in Aldor work pretty much as one would expect, except that they may be dependent.

```
RecordType : Type == Record(i:Integer, j:Boolean)

rr : Record(i:Integer, j:Boolean)
   == [i==4,j==true];
```

Raw Syntax

$t, T \in Term$	::=	$\ldots$	
		$\mid \mathtt{Record}(T_1, \ldots, T_n)$	record type
		$\mid$ `bracket` $\mid$ `record`	record introduction
		$\mid$ `apply`	field access
		$\mid$ `explode`	record elimination

`record` and `bracket` are synonyms. $\mathtt{bracket}(t_1, \ldots, t_n)$ can be written as $\mathtt{[}t_1, \ldots, t_n\mathtt{]}$. $\mathtt{apply}(t, x_i)$ can be written as $t\,\mathtt{.}\,x_i$.

Typing Rules

$$\frac{\Gamma \vdash (x_1 : T_1, \ldots, x_n : T_n) : \texttt{Tuple Type}}{\Gamma \vdash \texttt{Record}(x_1 : T_1, \ldots, x_n : T_n) : \texttt{Type}} \text{ record form}$$

$$\frac{\Gamma \vdash \texttt{Record}\overrightarrow{T} : \texttt{Type}}{\Gamma \vdash \texttt{bracket} : \overrightarrow{T}\texttt{->Record}\overrightarrow{T}} \text{ record intro1}$$

$$\frac{\Gamma \vdash \texttt{Record}\overrightarrow{T} : \texttt{Type}}{\Gamma \vdash \texttt{record} : \overrightarrow{T}\texttt{->Record}\overrightarrow{T}} \text{ record intro2}$$

$$\frac{\Gamma \vdash \texttt{Record}\overrightarrow{T} : \texttt{Type}}{\Gamma \vdash \texttt{explode} : \texttt{Record}\overrightarrow{T}\texttt{->}\overrightarrow{T}} \text{ record elim1}$$

$$\frac{\Gamma \vdash r : \texttt{Record}(x_1 : T_1, \ldots, x_n : T_n)}{\Gamma \vdash \texttt{apply}(r, x_i) : T_i[x_i := r.x_1, \ldots, x_{i-1} := r.x_{x-1}]} \text{ record elim2}$$

- Here type tuples are useful! E.g. note that the formation rule above allows for dependent records. The introduction rules rely on the keyword argument style for function application.
- Note that `record`, `bracket`, `explode`, and `apply` are heavily overloaded functions. Maybe it would be better not to do so in the formalisation?
- Using an enumeration type $'x_i'$, `apply` can be regarded as a function

  $$\frac{\Gamma \vdash \texttt{Record}(x_1 : T_1, \ldots, x_n : T_n) : \texttt{Type}}{\Gamma \vdash \texttt{apply} : (\texttt{Record}(x_1 : T_1, \ldots, x_n : T_n), 'x_i')\texttt{->}T_i}$$

  and $\texttt{apply}(r, x_i)$ can then be seen as a normal application, as explained in the Aldor User Guide (pages 146–147). However, this is only correct for non-dependent records; For dependent records a substitution is needed in the result type T_i.
- There are two more operations on records: `set!` and `dispose` (see pages 146–147 of the Aldor User Guide). We don't consider these as they are imperative operations. (What is interesting about these operations is that they show that records are not values, but rather references to values.)
- The Aldor compiler rejects untyped definitions of records

  ```
  t == record(i==4,j==5);
  ```

  but will accept a definition of the form

  ```
  t == record(i:Integer==4,j:Integer==5);
  ```

- Some or all the x_i in a type $\texttt{Record}(x_1 : T_1, \ldots, x_n : T_n)$ can be omitted. In that case the corresponding `apply`'s are missing. So degenerated records such as $\texttt{Record}(T_1, T_2)$ are allowed in Aldor. Because all the `apply`'s are missing, this is effectively the same as $\texttt{Cross}(T_1, T_2)$[9]. Still, if T_1 and T_2 are equal, then Aldor rejects $\texttt{Record}(T_1, T_2)$.

- Again, the Aldor compiler treats `Record` as a function of type `Tuple Type->Type`. In the formalisation we choose not to do this, for the same reasons as for `Cross` and `Enumeration`.

12 Unions

Union types provide disjoint union, also known as variants. For example,

```
IntOrBool : Type = Union (left:Boolean, right:Integer);

leftBool  : Union (left:Boolean, right:Integer) == [left==true];

rightInt : Union (left:Boolean, right:Integer) == [right==5];
```

Raw Syntax

$$
\begin{array}{lcll}
t, T \in \mathit{Term} & ::= & \ldots & \\
 & & \mid \texttt{Union}\overrightarrow{T} & \text{union type} \\
 & & \mid \texttt{bracket} \mid \texttt{union} & \text{union introduction} \\
 & & \mid \texttt{case} & \text{union test} \\
 & & \mid \texttt{apply} & \text{union elimination}
\end{array}
$$

`union` and `bracket` are synonyms. $\texttt{bracket}(t_1, \ldots, t_n)$ can be written as $\texttt{[}t_1, \ldots, t_n\texttt{]}$. $\texttt{apply}(t, x_i)$ can be written as $t\texttt{.}x_i$.

Typing Rules

$$\frac{\Gamma \vdash T_i : \texttt{Tuple Type}}{\Gamma \vdash \texttt{Union}(x_1 : T_1, \ldots, x_n : T_n) : \texttt{Type}} \text{ union form}$$

$$\frac{\Gamma \vdash \texttt{Union}(x_1 : T_1, \ldots, x_n : T_n) : \texttt{Type}}{\Gamma \vdash \texttt{bracket} : (x_i : T_i)\texttt{->Union}\overrightarrow{T}} \text{ union intro1}$$

$$\frac{\Gamma \vdash \texttt{Union}(x_1 : T_1, \ldots, x_n : T_n) : \texttt{Type}}{\Gamma \vdash \texttt{union} : (x_i : T_i)\texttt{->Union}\overrightarrow{T}} \text{ union intro2}$$

$$\frac{\Gamma \vdash \texttt{Union}(x_1 : T_1, \ldots, x_n : T_n) : \texttt{Type}}{\Gamma \vdash \texttt{case} : (\texttt{Union}\overrightarrow{T}, {}'x_i{}')\texttt{->Boolean}} \text{ union elim1}$$

$$\frac{\Gamma \vdash \texttt{Union}(x_1 : T_1, \ldots, x_n : T_n) : \texttt{Type}}{\Gamma \vdash \texttt{apply} : (\texttt{Union}\overrightarrow{T}, {}'x_i{}')\texttt{->}T_i} \text{ union elim2}$$

- Note that `record`, `bracket`, `apply`, and `case` are heavily overloaded functions. Especially the first three, as these are also used for records. (And again, maybe it would be better not to do so in the formalisation?)

[9] at least, as far as the functional sublanguage of Aldor is concerned; if imperative operations are taken into account, there are differences, as the components of a record can be imperatively updated.

- The rules for union types are not type-safe. The culprit is the elimination of union types (as usual). For example, if we define

  ```
  x : Union (left:Boolean, right:Integer) == union(right==5);
  ```

  there is nothing preventing us from considering `x` as a left-injection, as in

  ```
  unsafeProjection : Boolean == apply(x,left)
  ```

  So it's left up to the user to check – using the function `case` – that the correct component is extracted from a variant.

- There are two more operations on unions: `set!` and `dispose` (see p. 147/148 of the Aldor User Guide). We don't consider these as these are really only interesting in imperative setting. What is interesting about these operations is that they show that records are not values, but rather references to values.

13 Enumeration

Enumeration types in Aldor consist of a fixed collection of symbolic values. For example

```
Colour : Type == 'red,green,blue';

x: Colour  == red;
```

Raw Syntax

$$
\begin{array}{rcl}
t, T \in Term & ::= & \dots \\
& & \mid {}'x_1, \dots, x_n{}' \\
& & \mid \texttt{Enumeration}\overrightarrow{T}
\end{array}
$$

Typing Rules

$$\frac{}{\Gamma \vdash {}'x_1, \dots, x_n{}' : \texttt{Type}} \text{ enumeration form}$$

$$\frac{}{\Gamma \vdash x_i : {}'x_1, \dots, x_n{}'} \text{ enumeration intro}$$

- Aldor in fact regards ${}'x_1, \dots, x_n{}'$ as shorthand for

  $$\texttt{Enumeration}(x_1 : \texttt{Type}, \dots, x_n : \texttt{Type})$$

  Here `Enumeration` takes an arbitrary type tuple as argument, i.e.

  ```
  Enumeration : Tuple Type -> Type
  ```

  We choose not to do this in the formalisation. In addition to the problems with treating type tuples as first-class citizens, already discussed in Section 5.2, it is not clear what the meaning would be of `Enumeration` applied to a type tuple that is not of the form $(x_1 : \texttt{Type}, \dots, x_n : \texttt{Type})$.

- The Aldor compiler behaves strangely if we have overlapping enumeration types. This seems to be a bug. It would be better to disallow any overlap between enumeration types.
- How do clashes between enumerations and variables work? Eg. what if one of the x_i is also used as a variable?
- A difficulty with enumeration types is that in a "typeless" definition of the form x==x_i it may not be clear hard to tell that x_i is an element on an enumeration type, and which enumeration type. The `import` statement is used to declare enumeration types.

$$\frac{\Gamma \texttt{import from } {}'x_1,\ldots,x_n'; \Gamma' \vdash ok}{\Gamma; \texttt{import from } {}'x_1,\ldots,x_n'; \Gamma'; x\texttt{==}x_i \vdash ok} \text{ untyped definition ok}$$

The formalisation of the type system is surprisingly tricky here.

14 Equality

The type expressions in Aldor are complicated enough for equality of types to be non-trivial. There are different places where the notion of equality between types plays a role, and we can distinguish different notions of equalities between types. These are discussed below.

Notions of Equality

Different sources of equalities between types are

- *α-equality.*
  There are *bound variables* in types, so there is a notion of α-equality of types, i.e. equality up to renaming of bound variables. For example, types `(n:Integer)->Vector(n)->Integer` and `(m:Integer)->Vector(m)->Integer` can be regarded as equal.

  Related to α-equality is the case of vacuous dependency: one would expect that the types `S -> T` and `(x:S) -> T` would be equal in the case that `x` is not free in the result type `T`.
- *δ-equality.*
  We can define names for types, so there is a notion of δ-equality, i.e. equality up to (un)folding of definitions. For example, if we define `XX : Type == Integer` then the types `XX` and `Integer` can be regarded as equal.
- *β-equality.*
  Because types can contain lambda abstractions and applications in types there can be β-redices in type expressions. For example, the type `Integer` and the type `((X:Type):Type+->X) (Integer)` – the identity function on types applied to the type `Integer` – can be regarded as equal.

  In the same way one can consider η-equality as well as β-equality.
- Finally, because there are dependent types, types can have arbitrary terms as subexpressions. So any notion of equality for such sub-expressions induces a notion of equality on types. For example, because `3+4` and `7` are equal the types `Vector(3+4)` and `Vector(7)` can be regarded as equal.

It should be clear that a notion of equality that includes the equalities discussed under the last point above will quickly become undecidable. For instance, dependent types can contain diverging sub-expressions. In fact, just incorporating β-equality would be enough to make equality undecidable. The general problem with the last two notions of equality above is that type checking, which is done at compile time, becomes entangled with evaluation, e.g. of `3+4` to `7`, which by definition is done at run time. This is a well-known problem with dependent types, discussed for instance in [MR86].

Uses of Equality

There are several places where the type system depends on the notion of equality for types :

(i) Any inference rule where the same type occurs more than once in the premisses relies implicitly on a notion of equality. The most obvious place where this occurs is in the application rule

$$\frac{\Gamma \vdash f : S\texttt{->}T \qquad \Gamma \vdash s : S}{\Gamma \vdash f(s) : T} \text{ function elim}$$

Here the type S of the argument s has to be equal to the domain of f.

(ii) Less obvious than in the typing rule above, any inference rule where a type is required to of a particular form in one of the premisses also relies on a notion of equality, For example, in

$$\frac{\Gamma \vdash T : \texttt{Type}}{\Gamma \vdash \texttt{Tuple}\ T : \texttt{Type}}$$

type of T is required to be equal to `Type`. And, in the application rule again, the type of f is required to be equal to something of the form $S\texttt{->}T$.

(iii) Finally, overloading depends on equality – or rather, inequality – of types. Eg. the rule

$$\frac{\Gamma \vdash ok \qquad \Gamma \vdash t : T \qquad \neg(\Gamma \vdash x : T)}{\Gamma ; x\texttt{==}t : T \vdash ok}$$

requires that Γ does not contain any definition or declaration of an x of type T, nor of a type equal to T.

Ideally, in the formal description we would want to deal with equality by including a conversion rule of the form

$$\frac{\Gamma \vdash t : T \quad (T, T') \in R}{\Gamma \vdash t : T'} \text{ conversion}$$

where R is the equality relation on types. Intuitively, this rule states that we are only interested in the typing relation up to the notion of equality R on types. [10]

Unfortunately, this is not how equality is dealt with by the Aldor compiler. It turns out that the compiler uses several notions of equality, and uses different notion of equality in different places. An accurate description of Aldor can therefore not be given by including a single conversion rule as discussed above;

[10] Note that once δ-equality is included, the notion of equality R will depend on the context Γ, so we should really write $(T, T') \in R_\Gamma$ or $\Gamma \vdash (T, T') \in R$.

Instead equality would have to be build into any typing rule that relies on equality, e.g.

$$\frac{\Gamma \vdash f : S\texttt{->}T \qquad \Gamma \vdash s : S' \qquad (S, S') \in R}{\Gamma \vdash f(s) : T}$$

where R is the notion of equality used in this particular case.

We will not attempt to give an accurate description of equality in Aldor in this way: it would be very hard to do and not be very useful, since this is an aspect of the Aldor type system that we want to change anyway. Instead, we will make an inventory of the different notions of equality used in Aldor and give a rough indication of which notion of equality is used where.

Of the notions of equality listed earlier, the Aldor compiler only ever uses α- and δ-equality with some restrictions. In light of the difficulties that arise with the other notions of equality this is not surprising.

14.1 α-equality

Nearly everywhere the Aldor compiler treats α-equal types as being equal. So, in the formal description we could consider including

$$\boxed{\frac{t =_\alpha t'}{\Gamma \vdash (t, t') \in R}\ \alpha\text{-conv}}$$

There are only two case where the Aldor compiler does not work modulo α-equality:

- In a definition of a (dependently typed) function of the form

  ```
  f : (x:S)->T == (x:S):T +-> t
  ```

  the compiler insists that the same variable name `x` is used in the body `(x:S):T+->t` of the definition as in the type `(x:S)->`.

- The Aldor compiler does not always spot α-equality when checking for ambiguous overloading (as discussed under (iii) above). When the types involved become complicated the compiler may fail to spot that the same constant is defined twice for α-equal types, as for example `Id` in the definitions below:

  ```
  Id(X:Type,x:X) : X == x;
  Id(Y:Type,y:Y) : Y == y;
  ```

  It is interesting to note that when it comes to applications (as discussed under (i) above) then Aldor has no problems in spotting that the types `(X:Type,x:X)->X` and `(Y:Type,y:Y)->Y` are equal. So different algorithms for deciding equality for types are used when in comes to (i) and (iii).

14.2 δ-equality

Aldor treats typed definitions (of the form `x:T==t`) and untyped definitions (of the form `x==t` differently when it comes to definitional equality. It seems that we do not have δ-equality for the former but that we do have δ-equality for the latter, albeit in a limited form. This explains to some extent why typed and untyped definitions are treated differently in packages, as discussed in Section 7.

14.2.1 Definitions of the form `x : T == t`

The Aldor compiler does not use δ-equality for these definitions. So we do *not* have

$$\frac{\Gamma; X : \texttt{Type==}T; \Gamma' \vdash t : X}{\Gamma; X : \texttt{Type==}T; \Gamma' \vdash t : T}\ \delta \text{ unfold}$$

nor vice versa.

The only exception seems to be definitions of categories. Here the Aldor compiler does use δ-equality. This seems to contradict the Aldor User Guide, where on page 113/114 it is said that the `define` keyword has to be included, so that we have a definition of the form `define x :  T == t`, in order for to have δ-conversion for definitions of categories. We have

$$\frac{\Gamma; x : \texttt{Category==}d; \Gamma' \vdash t : x}{\Gamma; x : \texttt{Category==}d; \Gamma' \vdash t : d}\ \delta \text{ category unfold}$$

but we do *not* have the reverse, i.e.

$$\frac{\Gamma; x : \texttt{Category==}d; \Gamma' \vdash t : d}{\Gamma; x : \texttt{Category==}d; \Gamma' \vdash t : x}\ \delta \text{ category fold}$$

So the notion of equality that Aldor uses is not always symmetric!

All this suggests that as far as definitions of the form `x : T == t` are concerned, we only have

$$\boxed{\frac{(x : \texttt{Category==}d) \in \Gamma}{\Gamma \vdash (x, d) \in R}\ \delta \text{ category unfold}}$$

14.2.2 Definitions of the form `x == t`

It seems that in most cases the Aldor compiler works modulo δ-equality as far as these definitions are concerned. So

$$\boxed{\begin{array}{l} \dfrac{(x\texttt{==}t) \in \Gamma}{\Gamma \vdash (x, t) \in R}\ \delta \text{ untyped unfold} \\ \dfrac{(x\texttt{==}t) \in \Gamma}{\Gamma \vdash (t, x) \in R}\ \delta \text{ untyped fold} \end{array}}$$

The exception is that Aldor seems to ignore these equalities when it comes to spotting ambiguous overloading. For example, the Aldor compiler accepts the following definitions

```
XX == Integer;
five : Integer == 5;
five : XX == 6;
```

and does not complain that this overloading of `five` is ambiguous. (Here again, Aldor is better as spotting equality when it comes to (i) and (iii); when it comes to applications the compiler treats `XX` and `Integer` as equal.)

15 Subtyping, Courtesy Conversions, Satisfaction

Subtyping is a relation $\leq$ on types that comes with a so-called subsumption rule

$$\frac{\Gamma \vdash t : T \quad T \leq T'}{\Gamma \vdash t : T'} \text{ subsumption}$$

There are two – quite different! – possible semantics of subsumption:

- *apply some coercion function.*
  Maybe we have to apply some coercion function to convert a term of type T to type T'. For example, many languages treat the integers as a subtype of the reals, and here typically a coercion function has to be applied to convert integers to some floating-point format.

- *do nothing.*
  It may be the case that we don't have to do anything to a convert a term t of type T to get a term of type T'. Here one can think of subtyping between `Ring` and `Monoid`.

Note the similarity with the subsumption rule above and the conversion rule given on page 39. It might be hard to tell the two apart. The intuition behind them is quite different though, and the semantics of type conversion *has* to be "do nothing".

In nice type systems $\leq$ subsumes the notion of equality for types R, and R will even be equal to $\leq \cap \geq$. (In Aldor this is not true: e.g. there are courtesy conversions from cross products to multiple values and back, but these are different types.)

Aldor has 3 notions of "subtyping", which will be described in the subsections below, namely

- subtyping , $\Gamma \vdash S \sqsubseteq T$
- courtesy conversions , $\Gamma \vdash S \leq_{convert} T$
- satisfaction , $\Gamma \vdash S \leq_{sat} T$

It is not really clear what the differences between these three notions, and in how far we have to distinguish these notions in the formal description here. They may have different semantics, but that is not really an issue in the (syntactic) description of the type system.

15.1 Courtesy Conversions

We write $\Gamma \vdash S \leq_{convert} T$ for "there is a courtesy conversion from S to T". The Aldor User Guide (p. 84) lists the following rules for courtesy conversions:

$$\frac{}{\Gamma \vdash \texttt{Cross}(T, \ldots, T) \leq_{convert} \texttt{Tuple}\ T}$$

$$\frac{}{\Gamma \vdash \texttt{Cross}(T) \leq_{convert} T}$$

$$\frac{}{\Gamma \vdash T \leq_{convert} \texttt{Tuple}\ T}$$

$$\frac{}{\Gamma \vdash T \leq_{convert} \texttt{Cross}\ T}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \leq_{convert} T'}{\Gamma \vdash t : T'}\ \leq_{convert}\text{-subsumption}$$

- As the name suggests, the semantics of $\leq_{convert}$-subsumption involves coercion functions.
- Aldor does not provide any "congruence" rules to lift $\leq_{convert}$ to more complicated type expressions (like it does for $\sqsubseteq$). (Because of this, there is no need to include a reflexivity rule for $\leq_{convert}$; The only use of reflexivity of $\leq_{convert}$ would be in the subsumption rule, and there it's obviously not really needed.)
- In addition to courtesy conversions, Aldor also has "primitive conversions" and "conversion functions" (see p. 84-85 of the Aldor User Guide).
- There are also courtesy conversions between multiple values and tuples/cross products

  $$\frac{}{\Gamma \vdash (T, \ldots, T) \leq_{convert} \texttt{Tuple}\ T}$$

  $$\frac{}{\Gamma \vdash (T_1, \ldots, T_n) \leq_{convert} \texttt{Cross}(T_1, \ldots, T_n)}$$

  $$\frac{}{\Gamma \vdash \texttt{Cross}(T_1, \ldots, T_n) \leq_{convert} (T_1, \ldots, T_n)}$$

  but, as we do not consider multiple values, we ignore these. Observe that the last two courtesy conversions effectively render cross products and multiple values equivalent.

15.2 Subtyping

We write $\Gamma \vdash S \sqsubseteq T$ for S is a subtype of T in context Γ. The Aldor User Guide (p. 83) lists the following rules for subtyping:

$$\frac{m \geq n \quad p_1, \ldots, p_m \text{ permutes } 1, \ldots, m}{\Gamma \vdash \texttt{with}\{x_1 : T_1; \ldots; x_n : T_n\} \sqsubseteq \texttt{with}\{x_{p_1} : T_{p_1}; \ldots; x_{p_m} : T_{p_m}\}} \sqsubseteq\text{-width}$$

$$\frac{\Gamma \vdash S_2 \sqsubseteq S_1 \quad \Gamma \vdash T_1 \sqsubseteq T_2}{\Gamma \vdash S_1\texttt{->}T_1 \sqsubseteq T_1\texttt{->}T_2} \sqsubseteq\texttt{-->}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T \sqsubseteq T'}{\Gamma \vdash t : T'} \sqsubseteq\text{-subsumption}$$

- Note that in the reordering of the items in a signature according to the permutation $p_1, \ldots, p_m$ it is assumed that the permuted signature is still valid; that this is not always the case is a consequence of type dependency.
- The semantics of $\sqsubseteq$-subsumption is "do nothing": as explained in the Aldor User Guide (p. 83), if $T \sqsubseteq T'$ then they share an underlying "base domain" and their elements have the same representation.
- According to the manual, the rules $\sqsubseteq$-width and $\sqsubseteq$`-->`are bi-implications, e.g.

  $$S_1\texttt{->}T_1 \sqsubseteq T_1\texttt{->}T_2 \;\Rightarrow\; S_2 \sqsubseteq S_1 \wedge T_1 \sqsubseteq T_2$$

  Of course, if the rules above are the only rules for subtyping then this is clearly true.
- Observe that this is a limited notion of 'width' subtyping. It is not possible to subtype on a particular field – that is to allow $x_i : T_i'$ to replace $x_i : T_i$, where $T_i \sqsubseteq T_i'$ – in moving from subtype to supertype; this is known as 'depth' subtyping. Subtyping on fields of records is also not permitted.

15.3 Satisfaction

We write $\mathit{\Gamma} \vdash S \leq_{sat} T$ for S satisfies T in context $\mathit{\Gamma}$. The Aldor User Guide (p. 86) lists the following rules for satisfaction:

$$\frac{\mathit{\Gamma} \vdash S : \texttt{Category}}{\mathit{\Gamma} \vdash S \leq_{sat} \texttt{Type}}$$

$$\frac{}{\mathit{\Gamma} \vdash \texttt{Category} \leq_{sat} \texttt{Type}}$$

$$\frac{\mathit{\Gamma} \vdash \texttt{add}\{\ldots\} : S \quad \text{S is the type of a category or a domain}}{\mathit{\Gamma} \vdash S \leq_{sat} \texttt{Type}}$$

$$\frac{\mathit{\Gamma} \vdash S \sqsubseteq T}{\mathit{\Gamma} \vdash S \leq_{sat} T} \ \sqsubseteq \Rightarrow \leq_{sat}$$

$$\frac{\mathit{\Gamma} \vdash S \leq_{convert} T}{\mathit{\Gamma} \vdash S \leq_{sat} T} \ \leq_{convert} \Rightarrow \leq_{sat}$$

$$\frac{\mathit{\Gamma} \vdash t : T \qquad \mathit{\Gamma} \vdash T \leq_{sat} T'}{\mathit{\Gamma} \vdash t : T'} \ \leq_{sat}\text{-subsumption}$$

- Note that satisfaction subsumes the courtesy conversions.
- The User Guide also gives a rule

$$\frac{\mathit{\Gamma} \vdash T : \texttt{Type}}{\mathit{\Gamma} \vdash T \leq_{sat} ()}$$

  It is not clear what is the intended meaning of the type () here.
- There are also satisfactions involving `Exit`:

$$\frac{\mathit{\Gamma} \vdash T : \texttt{Type}}{\mathit{\Gamma} \vdash \texttt{Exit} \leq_{sat} T}$$

- The User Guide gives the rule

$$\frac{\mathit{\Gamma} \vdash C\texttt{add}\{D\} : S}{\mathit{\Gamma} \vdash S \leq_{sat} \texttt{Category}}$$

  The judgement '$\mathit{\Gamma} \vdash C\texttt{add}\{D\} : S$' is intended to formalise 'S is the type of a category (in the context $\mathit{\Gamma}$)', but isn't `Category` the only possible type of a category?
- The User Guide gives the rule

$$\frac{\mathit{\Gamma} \vdash S\texttt{with}\{D\} : T}{\mathit{\Gamma} \vdash T \leq_{sat} \texttt{Type}}$$

  together with

$$\frac{\mathit{\Gamma} \vdash T \leq_{sat} \texttt{Category}}{\mathit{\Gamma} \vdash T \leq_{sat} \texttt{Type}}$$

  but again, isn't `Category` the only possible type of a category?

16 Omissions

The formalisation of the Aldor type system outlined in this report has deliberately omitted various aspects of the system. As was argued in the introduction, this is for a variety of reasons, but principally because our aim is to formalise what is – at least from a type-theoretic point of view – the *essence* of Aldor. The remainder of this section surveys aspects which are covered either partially or not at all.

Aldor is an imperative language, with a functional core, much in the mould of Standard ML [MTHM97]. We have confined our attention to the functional subset in this report, but that is an inessential restriction. (Readers might doubt this because references cause subtle problems for the type system of SML, but this is due to the interaction of references and parametric polymorphism, which is absent from Aldor.)

We have not, on the whole, discussed questions about the scope of definitions, since these are largely orthogonal to type checking and inference. They do interfere when default argument values are allowed, and also when type checking (mutually) recursive definitions. However, once scopes are resolved the type checking issues are relatively straightforward. Scopes are also controlled by means of import and export statements; we have only covered the fundamentals of the `import` mechanism. *Post facto* extensions also affect scopes of packages; again these are not covered here.

A related difficulty comes with arguments passed to functions by keyword. These break the usual property that functions are independent of the names of their bound variables (the property of α-conversion), and so break the property that the interface of a function is entirely specified by its type.

For example, we would normally treat the definitions

```
id(n : Integer) : Integer == n ;
```

and

```
id(m : Integer) : Integer == m ;
```

as defining the same (identity) function over `integer`. However, with keyword arguments, the application

```
id(n == 7)
```

is a well-formed application of the first definition of `id` but *not* of the second. As was said earlier, in order to apply the function `id` we need to know not only the types of the arguments but also their names, and so the latter information forms part of the interface to the function.

Aldor contains a plethora of notions of 'product' or 'tuple' types. We have, in particular, not covered multiple values. As was discussed in the body of the report, we have also chosen to treat record formation and related operations as primitives, rather than as applications of functions to type tuples; this is discussed again in Section 18.

Because of their nature, Aldor macros are independent of the type system. Observe, however, that the treatment of ADTs does not treat `rep` and `per` as macros but instead uses a scoping mechanism to type check their application.

Much of the description of Aldor in the manual involves defining many different sorts of expression; as was remarked earlier many of these operations are variants of function application, which is covered in detail in this report.

Categories can be built in a structured way, either by extension using `with` or by putting together two signatures with `join`. A suitable expansion prior to type checking means that we do not deal with these forms; on the other hand, this expansion approach precludes our dealing with variables which range over categories; rather we assume that definitions can be fully expanded whenever that proves to be necessary.

In examining dependencies between fields of a package we have not allowed for mutual dependencies in our rules; this can be accommodated by standard means. Dependencies between the fields of a `Cross` product are also allowed by Aldor; it is by no means clear how these constructions are used.

17 Aldor compiler errors

At various points in the report we have noted what appear to be errors in the version 1.1.10b of the Aldor compiler; it might be that these have been fixed in later releases, or that indeed they are 'features' rather than errors. We list them here, giving links back into the body of the report where appropriate.

- The compiler does not always treat ambiguous definitions in the same way; this was discussed in Section 4, page 14.
- The compiler has problems with dependent types as first-class citizens, as noted in Section 6.2, page 20.
- The compiler has problems with dependent packages and crashes when these become complicated; see Section 7.1.4, page 28.
- The compiler behaves strangely if we have overlapping enumeration types; see Section 13, page 38.

Other aspects of the language implementation are less serious than these, but certainly contravene the description of the language in the User Guide, [WBD+94].

- The keyword `define` is supposed to make a definition (of a category) transparent ([WBD+94], p113); in fact it appears to have no effect on the way in which the definition is interpreted. There are also important differences between the two definition forms `x == t` and `x:T == t` which are not apparent from [WBD+94].
- The compiler can crash when the same name is used for two fields in a (dependent) record; it is not clear whether this is intended or not, but it is not a feature that would be put to heavy use by the average user.

18 Recommendations

In the light of examining the language and its type system we have come to various conclusions about how its design might be improved. A number of these suggestions would simplify the language; others would combine features and a third class suggests extending the language in various natural ways.

Type tuples

One can wonder if Aldor does not go a bit too far in treating everything as first-class citizens.

For example, Aldor treats `Record` (and similarly `Cross`, `Enumeration`, `Union`, etc.) as a first-class citizen, namely as a function of type

```
Record : Tuple Type->Type
```

This requires type tuples to be treated as first-class citizens, so that the can be passed as arguments to a function such as `Record`. On the one hand, this very compact typing of `Record` is very appealing. But on the other hand it causes some problems.

The most serious problem is that if type tuples are first-class citizens, then we can have variables `X:Tuple Type`, and hence record types `Record X` of which the fields cannot be known at compile time.

Another problem is that the typing of `Record` above is somewhat imprecise, as it does not impose any restrictions on the kind of type tuples that `Record` can get as an argument. Recall that type tuples can be of the form $(T_1, \ldots, T_n)$, or of the form $(x_1 : T_1, \ldots, x_n : T_n)$, or any combination of the two. The typing of `Record` above leaves open the question whether `Record` can for instance be applied to `(Integer,Integer)`, and, if so, what the meaning of `Record(Integer,Integer)` might be, as this record type does not contain any field names.

Because of these problems in the formalisation we have chosen to treat `Record`$(x : T_1, \ldots, x_n : T_n)$ as a primitive term construction, and not the application of a function of `Record` to the type tuple $(x : T_1, \ldots, x_n : T_n)$.

Similarly, one could wonder if there are not more places where constructions should be regarded as primitives rather than as applications, for instance `record`.

Over-generality

The Aldor compiler allows a number of things which do not seem to make sense. For instance, the compiler does not complain if we give it a package without definitions or even names for fields,

```
sillyPackage == add {x:Integer;y:Integer} ;
anotherSillyPackage == add {Integer;Integer} ;
```

or cross product types with definitions for fields,

```
sillyCrossProduct : Type
                == Cross (Integer,x:Integer==7);
```

packages written with `,` instead of `;`

```
sillyPackage == add {x:Integer,y:Integer} ;
```

and many more. These are all things that could – and should – be detected already at the parsing stage by the compiler, i.e. before typing is considered. The fact that it is not maybe because the – very general – notion of type tuple is used here.

Definition forms

We suggest that there should be one form of definition, namely a transparent definition. Specifically, given the definition

```
x : T == t
```

both the type `T` and the value `t` of the name `x` should be visible within its scope. Note that normally there are contexts in which it is sensible to reveal only the type of a name; the presence of dependent types in Aldor makes it necessary to have access to the value more often than in other languages.

This being said, there is still an opaque definition mechanism, namely the ADT mechanism, and this can be used when abstraction is wanted.

Product and tuple types

Aldor contains various different notions of 'container' type: multiple values, cross products, tuples, lists and records. It should be possible to rationalise these into a number of different constructions with different purposes.

- Lists – or tuples in Aldor-speak – can be used to form homogeneous finite collections of values.
- Cross products can be used to form heterogeneous combinations of fixed size; records provide a named variant of these.

Macros

Macros can be removed from the language. The advantage of such a move would be to bring all of the language under the type checker; this is not currently the case because macro expansion takes place before type checking.

Macros are ostensibly used to support the ADT implementation, but we have shown in Section 7.2 that this can be done without using macros.

Another use suggested by [WBD+94], Section 12.4, is the use of a particular macro definition like

```
li? x ==> (not empty? x and empty? rest x)
```

over more than one type. This effectively mimics parametric polymorphism using macros; it can be replaced by a function in which the type of the list is passed in as an explicit parameter

```
li? (T:Type,x:List(T)) : Boolean == (not empty? x and empty? rest x)
```

and this definition is now susceptible to type checking when it is used.

Dependent types

As we have argued elsewhere, [PT98], the dependent types of Aldor should be implemented in such a way that type expressions are evaluated, equating, for instance, vectors of length `2+3` and vectors of length `5`. This modification is a focus of current work at the University of Kent; further details are available at

`http://www.cs.ukc.ac.uk/people/staff/sjt/Atypical/`

The more general aspect of equality in Aldor is examined next.

Equality

It should be possible to simplify the treatment of equality in Aldor, which Section 14 shows is currently tricky. We would argue that there should be a single notion of equality in Aldor, under which values – including types – are evaluated before being compared for identity of their fully-evaluated (or 'normal') forms.

This works, except for the treatment of abstract data types. Consider the definition

```
newType : Type = add { ... }
```

This definition has two purposes: it is *definitive* in that it defines the value of `newType` but it is also *generative* in generating a new type named `newType`. The language Modula-3 adopts a similar approach to types, and there is an illuminating discussion of the rationale for this, '*How the types got their identity*', in Section 8.1 of [Nel91].

Subtyping

It should be possible to define a single notion of subtyping. If the system of 'container' types is simplified then this should make courtesy conversions substantially simpler. Once this is achieved, it will be possible to define a single notion of (width and depth) subtyping as alluded to in Section 15.

Additional features

Some obvious things are missing from Aldor, notably

- mutually abstract datatypes, in which the carrier types of two or more ADTs are mutually visible, and
- algebraic datatypes as in modern functional languages like SML and Haskell.

Miscellaneous points

Finally there are some miscellaneous points.

Lumping together packages and ADT's as one big collection of so-called domains is less than ideal. It would be better to leave out the packages and take ADTs equivalent to domains; alternatively one could treat packages as modules, a collection of entities quite separate from ADTs.

As noted in Section 12, the elimination rule for unions is type unsafe in that it is possible to treat a value of one 'variant' as if it belongs to another of a different type.

It should be possible to clarify the mechanism of keyword arguments and default values within the type system.

It would align Aldor with other functional languages if function application were made *left* associative.

19 Conclusions

The report has covered the essence of the Aldor type system and has shown that it can be explained by means of a compact set of type inference rules. A side-effect of the activity has been to point out some difficulties with the design of the type system, as well as some potential bugs in the implementation.

References

[Aug98] Lennart Augustsson. Cayenne – a language with dependent types. ACM Press, 1998.

[Bar93] Henk Barendregt. Lambda calculi with types. In Samson Abramsky et al., editors, *Handbook of Logic and Computer Science, Volume 2*. Oxford University Press, 1993.

[BR95] Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice & Experience*, 25(8):863–889, 1995.

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

[DD85] James Donahue and Alan Demers. Data types are values. *ACM TOPLAS*, 7(3):426–445, 1985.

[DTP99] Proceedings of the Workshop on Dependent Types in Programming. `http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99/proceedings.html`, 1999.

[How80] William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*. Academic Press, 1980. A reprint of an unpublished manuscript from 1969.

[LB88] Butler Lampson and Rod Burstall. Pebble, a kernel language for modules and abstract data types. *Information and Computation*, 76, 1988.

[Luo89] Zhaohui Luo. ECC, the Extended Calculus of Constructions. In *Logic in Computer Science*, pages 386–395. IEEE, 1989.

[Mac86] David MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*. ACM Press, 1986.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.

[ML79] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science*, volume VI, pages 153–175. North Holland, 1979.

[MR86] Albert R. Meyer and Mark B. Reinhold. 'type' is not a type: Preliminary report. In *POPL*, pages 287–295. ACM, 1986.

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

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[PT98] Erik Poll and Simon Thompson. Adding the axioms to Axiom: Towards a system of automated reasoning in Aldor. In *Calculemus and Types workshop*, July 1998. Also as Technical Report 6-98, Computing Laboratory, University of Kent.

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

[Ryd98] Chris Ryder. Report on the Aldor compiler. Computing Laboratory, University of Kent, 1998.

[San95] Philip S. Santas. A type system for computer algebra. *Journal of Symbolic Computation*, 19(1–3):79–110, 1995.

[Tho91] Simon Thompson. *Type Theory and Functional Programming*. International Computer Science Series. Addison-Wesley, 1991.

[Tou98] E. Touratier. *Etude de typage dans le systèe de calcul scientifique Aldor*. PhD thesis, Université de Limoges, 1998.

[WBD+94] 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.