Dynamic Types and Type Dependent Functions

Marco Pil

Computing Science Institute, University of Nijmegen,
Postbus 9010, 6500 GL Nijmegen, The Netherlands
marcop@cs.kun.nl

Abstract. When programs communicate with other programs, flexibility is demanded. Programs do not necessarily have information about each other. When assigning types to these communications a certain amount of dynamic typing is unavoidable. But we do not want our entire language to become dynamically typed, and consequently an interface between the statically and dynamically typed parts of the program has to be defined. Such an interface, using *dynamics*, has been introduced by Abadi et al. Leroy and Mauni extended the system of dynamics to allow the inclusion of polymorphic objects in dynamics. In this paper we extend the system even further with a restricted form of type dependent functions, which allow us to abstract over the types of the dynamics on functional level. In CLEAN, these type dependent functions will be implemented by overloading.

1 Introduction

When programs communicate with other programs, or possibly with other incarnations of themselves, flexibility is demanded. Programs do not necessarily know about each other's existence and may have been terminated before others are launched. Well-known examples of such communication are asynchronous message passing and file I/O.

Typing file I/O differs from typing ordinary expressions because at compile-time there is no knowledge about the contents of a file at run-time. In general, there is no static link between the name of a file and the type of its contents. Thus, types of the parts of a program that perform file I/O can only be checked at run-time, when reading or writing. Therefore a certain amount of dynamic typing is unavoidable.

But we do not want our entire language to become dynamically typed. Statically typed programming languages allow earlier error checking, and generation of more efficient object code than languages where all type consistency checks have to be performed at run-time. An interface between the statically and dynamically typed parts of the program has to be defined.

The concept of *objects with dynamic types*, or *dynamics* for short, as introduced by Abadi et al. [1], can provide such an interface. Values of this Dynamic type are, roughly speaking, pairs of a value and a type, such that the value component can be typed with the type component.

In Leroy and Mauni [7] the system of dynamics has been extended to be able to include polymorphic objects in dynamics. In this paper we extend the system with a restricted form of type dependent functions. These functions enable us to abstract over the types that are used in dynamic type patterns and in the construction of dynamic objects.

This paper has the following structure. First we will introduce dynamics in CLEAN, then we define a simple language with dynamic types. Thereafter we specify a type system for this language in which we work out the details: we will define an operational semantics, and prove decidability of type assignment.

2 Dynamics

This section introduces dynamics as they will be implemented in Dynamic [6], using the syntax as introduced in Pil [8].

For typing communication between independent programs, we would like to perform run-time type checks, but we do not want every object in the program to be tagged with its type. That is why we distinguish between objects with only a static type and objects with a dynamic type.

A dynamic consists of a normal object that has been paired with an encoding of its type to enable run-time type checks.

From a statical point of view all dynamics belong to same static type — the type Dynamic. Values of type dynamic are self-described as far as types are concerned. They contain enough type information to be type checked at run-time.

A dynamic is created by pairing an object with an encoding of its type, using the dynamic construct:

```
dynamic expr :: type
```

where it can be checked statically that the expression expr is indeed of the static type type.

Example 1. Examples of dynamics are:

```
dynamic True :: Bool
dynamic fib :: Int → Int
dynamic reverse :: [a] → [a]
```

The access to objects of type Dynamic is limited. The value can only be accessed if its type matches a specific statically determined type. For this purpose the pattern match mechanism of CLEAN has been extended to describe matching on types.

It is guaranteed at compile-time that if the type contained in the dynamic matches the statically specified type described in the patterns, then the value contained in the dynamic can safely be used in the right hand side expression, since it is known to be of the appropriate type.

Example 2. An example of a dynamic pattern match is:

```
f :: Dynamic → Int
f (x :: Int)        = x
f (g :: Int → Int) = g 5
f  else             = 0
```

The type patterns need not fully specify the demanded type: they may include *type pattern variables*, which match any subexpression of the dynamic's type. The type pattern has scope over the whole alternative, that is, over both the type pattern(s) and the right hand side expression.

Example 3. The use of type pattern variables allow the specification of powerful functions that are polymorphic in the types of the dynamics:

```
dynamicApply :: Dynamic → Dynamic → Dynamic
dynamicApply (f :: a → b) (x :: a) = dynamic (f x) :: b
dynamicApply  df            dx       = dynamic "Error" :: String
```

The new feature presented in this paper is that the type pattern variables do not need to have a binding occurrence in a type pattern. It can also be bound by the function type. In this case they scope over the whole function, i.e., the function type and all function alternatives (both left-hand side and right-hand side).

If a function type binds a type pattern variable a, this is indicated by adding a type context TC a and annotating the variables with a caret (^). The type class TC stands for *Type Code*, since the function has been overloaded in the representation of the type of a.

Example 4. A type context to bind type pattern variable a:

```
f :: Dynamic → a → Dynamic | TC a
f (g :: a^ → b) x = dynamic (g x) :: b
f  dg            x = dynamic "Error" :: String
```

At run-time it will be checked whether the dynamic argument of f is a function from some a to some b, and that this a matches the type for x that has been determined by the context in which f has been applied.

The type variables that are bound by a type context do not necessarily have to occur in dynamic type patterns. They can also appear in the dynamic type expressions on the right-hand side:

Example 5. The function wrap wraps a value into a dynamic:

```
wrap :: a → Dynamic | TC a
wrap x = dynamic x :: a^
```

The mechanism of overloading takes care that a correct type code is deduced from the context in which wrap has been used.

So type dependent functions allow us to abstract over types of dynamics beyond the scope of a function, just as type pattern variables let us abstract over the these types within one function alternative. This more powerful form of abstraction is often desired. For illustration, consider the function lookup that searches through a list of dynamics until it finds an integer and returns it, or fails to come up with such element and aborts.

```
lookup :: [Dynamic] → Int
lookup [ ]           = abort "No integer in the list"
lookup [x::Int : xs] = x
lookup [  dy   : xs] = lookup xs
```

Note that instead of allowing the program to abort, one can write a function lookup :: [Dynamic] → (Maybe Int) that yields No if the list contains no integer.

The lookup function can easily be altered to search the list for the first Real or Bool, but it cannot be generalised to a polymorphic function:

```
lookup :: [Dynamic] → a        (Wrong)
lookup [ ]         = abort "No element of the correct type in the list"
lookup [x::a : xs] = x
lookup [ dy  : xs] = lookup xs
```

This version of lookup is incorrect: because it is parametric in the representation of the type it is looking for, it is not actually polymorphic.

With the new mechanism of type dependent functions we are now able to write such general lookup function:

```
lookup :: [Dynamic] → a   | TC a
lookup [ ]          = abort "No element of the correct type in the list"
lookup [x::a^ : xs] = x
lookup [ dy   : xs] = lookup xs
```

where the actual type of a is determined by the context in which lookup has been used, for example (let dynlist be a list of dynamics):

```
(lookup dynlist) + 5

sin (lookup dynlist)
```

In the first example the first integer is picked from the dynlist and added to 5. The latter example applies the *sin* function to the first Real in the dynlist.

2.1 Implementation Issues

It is undesirable for the compiler to generate an specialised instantiation for all different types. Rather than ad-hoc polymorphism one would like parametric polymorphism; the same implementation for every instance.

To accomplish this, the CLEAN compiler will transform the type dependent functions, giving them extra arguments. The type (expressions) on which the function depends are passed in these arguments.

Example 6. The transformed version for the lookup function will be:

```
lookup :: (TypeCode a) → [Dynamic] → a
lookup tc [ ]          = abort "No element of the correct type in the list"
lookup tc [x::tc : xs] = x
lookup tc [ dy  : xs]  = lookup tc xs
```

And the applications of lookup will be transformed by the compiler to:

```
(lookup TypeCode_Int dynlist) + 5

sin (lookup TypeCode_Real dynlist)
```

The formal language in the next chapter is based on this idea; the fact that a function depends on a certain type is reflected by using an extra argument in which the type code can be passed.

3 A Formal System for Expressions with Dynamic Types

This section describes a formal system that formalizes the notion of objects with dynamic types. It is based on the formal system for term graph rewriting as introduced by Barendsen and Smetsers [3]. The formal system allows reasoning about decidability of typing and provides a handle for the presentation of the operational semantics.

First the syntax of the formal language will be given in full, and then the new constructs (w.r.t. the language introduced by Barendsen and Smetsers [3]) will be discussed individually. Finally typing rules will be given for the new constructs.

3.1 Syntax

Definition 1. *Types are built up from type variables, and type constructors:*

$$\sigma \ ::= \ \alpha \mid a \mid \mathsf{T}\ \vec{\sigma} \ \mid \sigma_1 \to \sigma_2$$

where T *ranges over the type constructors. The function type can be seen as a special binary type constructor.*

Note that there are two kinds of variables: generic variables $\alpha, \beta, \ldots$ that are used in type schemes, and type pattern variables $a, b, \ldots$ that are used for the construction and pattern matching of dynamics. Intuitively the generic variables α may be seen as universally quantified, while the type pattern variables a have the properties of existentially quantified variables.

Definition 2. *Expressions are generated by:*

$$
\begin{array}{rl}
E ::= & x \mid \mathbf{C}(E_1,\ldots,E_k) \mid \mathsf{let}\ x = E\ \mathsf{in}\ E' \mid \mathsf{letrec}\ \vec{x} = \vec{E}\ \mathsf{in}\ E \\
& \mid \mathsf{case}\ E\ \mathsf{of}\ \vec{P} = \vec{E} \mid E\ E' \mid \mathbf{F}(\sigma_1,\ldots,\sigma_j,E_1,\ldots,E_k) \\
& \mid \langle \sigma, E \rangle \\
& \mid \underline{\mathsf{open}}\ E\ \underline{\mathsf{as}}\ \langle a, x \rangle\ \underline{\mathsf{in}}\ E' \\
& \mid \underline{\mathsf{typecase}}\ a\ \underline{\mathsf{of}} \vec{P_t} = \vec{E} \quad \underline{\mathsf{fail}} = E' \\
& \mid \underline{\mathsf{equal}}\ a\ a' \quad \underline{\mathsf{success}} = E \quad \underline{\mathsf{fail}} = E' \\
& \mid \underline{\mathsf{coerce}}\ a\ a' \quad \underline{\mathsf{success}} = E \quad \underline{\mathsf{fail}} = E' \\
P ::= & \mathbf{C}(x_1,\ldots,x_k) \\
P_t ::= & \mathsf{T}\ a_1,\ldots,a_k
\end{array}
$$

where $\mathbf{C}$ ranges over data constructors, $\mathbf{F}$ over function symbols and T over the type constructors.

Definition 3. *The notion of* symbol type *has been redefined*[1] *to incorporate abstractions over type variables. A* symbol type *of arity k and type arity j is a quantified $k+1$ tuple $\Pi a_1 \ldots a_j.(\sigma_1,\ldots,\sigma_k,\tau)$. This will be denoted as:*

$$\Pi a_1 \ldots a_j.\ (\sigma_1,\ldots,\sigma_k)\ \triangleright\ \tau$$

The types $\vec{a}$ are called type arguments*, whereas the types $\vec{\sigma}$ are called* argument types*. The type τ is the* result type.

3.2 Introduction of New Constructs

Dynamics can be constructed by pairing a type and an expression:

$$\langle \sigma, E \rangle$$

where the type σ may include both type variables, to express polymorphism, and type pattern variables, which have been bound by an open or a typecase construct.

A dynamic can be unpaired using an open construct. Its value component is bound to a term variable (x) and its type component to a type pattern variable (a):

$$\underline{\mathsf{open}}\ \mathit{DynamicExpr}\ \underline{\mathsf{as}}\ \langle a, x \rangle\ \underline{\mathsf{in}}\ \mathit{Expr}$$

Now the actual type component should unify with a statically specifed type (or with another actual type component). For this there are three constructs: the typecase, the equal, and the coerce constructs.

The typecase-construct enables matching on type constructors.

$$
\begin{array}{ll}
\underline{\mathsf{typecase}}\ a\ \underline{\mathsf{of}}\ T_1\ \vec{a_1} & = E_1 \\
\qquad\qquad \vdots & \\
\qquad\qquad T_n\ \vec{a_n} & = E_n \\
\qquad\qquad \underline{\mathsf{fail}} & = E_{fail}
\end{array}
$$

[1] as compared to the definition in Barendsen and Smetsers [3]

The intended meaning is that the selector type-expression a is matched against the type patterns $T_i\ \vec{a_i}$. The first pattern $T_j\ \vec{a_j}$ that matches, will cause the branch E_j to be taken. If none of the patterns match, the branch E_{fail} will be taken.

The type pattern variables $\vec{a_i}$ are bound by the typecase, and may be used in E_i in the same way as variables that have been introduced by an open-construct.

As with the normal case-construct, we only consider *left-linear* functions: no type variable may occur more than once in the same pattern. Moreover, these type pattern variables are 'local': they only occur in $T_i\ \vec{a_i} = E_i$.

The equal-construct is used to compare two types at run-time.

$$\begin{array}{l} \underline{\mathsf{equal}}\ a\ b\ \underline{\mathsf{success}} = E_{succ} \\ \qquad\qquad\ \ \underline{\mathsf{fail}} \qquad = E_{fail} \end{array}$$

The intended meaning is that if a is unifiable b then the expression E_{succ} will be evaluated, otherwise E_{succ}.

However, it is not always desirable to unify types. The comparison can also consist of a coercion, to check whether one type is an instance of the other.

$$\begin{array}{l} \underline{\mathsf{coerce}}\ a\ b\ \underline{\mathsf{success}} = E_{succ} \\ \qquad\qquad\quad \underline{\mathsf{fail}} \qquad = E_{fail} \end{array}$$

With the intended meaning that if a coercible to b, i.e. if b is a instance of a, then the expression E_{succ} will be evaluated, otherwise E_{fail}.

Note that the constructs typecase, equal, and coerce match on type pattern variables instead of full types. This is done to make the proof of decidability of type checking easier (see Sect. 3.4), since more complex types are not required when CLEAN expressions are translated to this formal language.

The **type dependent functions** are represented on the level of function definitions by the fact that the first j arguments of a function are type pattern variables, and that the symbol type of such function quantifies over those j variables: $\Pi a_1 \ldots a_j.\ \vec{\sigma}\ \triangleright \tau$ (where the $\vec{a}$'s may occur in both $\vec{\sigma}$ and τ).

$$\mathbf{F}\ a_1 \ldots a_j\ x_1 \ldots x_k = E$$

The syntax of the **function application** has been adapted to allow the type arguments to the function to be specified:

$$\mathbf{F}(\sigma_1, \ldots, \sigma_j, E_1, \ldots, E_k)$$

where $\mathbf{F}$ has arity k and type arity j.

3.3 Rules for Type Assignment

In this section, after a few auxiliary definitions, the typing rules for the new language constructs will be given. The typing rules for the existing parts can be found in Barendsen and Smetsers [3].

Auxiliaries In addition to the standard basis ($\mathbb{B}$), we need a *type basis* ($\mathbb{T}$) that keeps track of which type pattern variables have been declared in surrounding expressions, and an *equation basis* ($\mathbb{E}$) that holds equations (of the form $\sigma \sim \tau$) of types that may be assumed to be equal but are syntactically different.

Definition 4. *(i) A term basis $\mathbb{B}$ is finite set of term variable declarations of the form $x : \sigma$, where x is term variable and σ is a type.*

(ii) A type basis $\mathbb{T}$ is finite set of type variable declarations of the form $\alpha : \star$, where α is a type (pattern) variable.

(iii) An equation basis $\mathbb{E}$ is a finite set of type equations of the form $\sigma \sim \tau$, where σ and τ are types.

(iv) We write Γ as an abbreviation of $\mathbb{B}, \mathbb{T}, \mathbb{E}$.

Definition 5. *(i) The set of type variables $\mathsf{TV}(\sigma)$ of a type σ is defined inductively by:*

$$\begin{array}{ll} \mathsf{TV}(\alpha) = \{\alpha\} & \mathsf{TV}(\mathsf{T}\ \vec{\sigma}\ ) \quad = \bigcup_i \mathsf{TV}(\sigma_i) \\ \mathsf{TV}(a) = \{a\} & \mathsf{TV}(\sigma_1 \to \sigma_2) = \mathsf{TV}(\sigma_1) \cup \mathsf{TV}(\sigma_2) \end{array}$$

(ii) The set of type variables $\mathsf{TV}(\mathbb{B})$ of a basis $\mathbb{B}$ is defined by naturally extending the definition:

$$\mathsf{TV}(\mathbb{B}) = \bigcup \{\mathsf{TV}(\sigma) \mid x : \sigma \in \mathbb{B}\}$$

Definition 6. *A type σ is* well-formed *w.r.t. a type basis $\mathbb{T}$*

$$\mathbb{T} \vdash \sigma : \star$$

iff it only contains variables that have been declared in $\mathbb{T}$, i.e. iff it can be constructed using the following rules:

$$\mathbb{T} \vdash \alpha : \star \qquad \mathbb{T}, a{:}\star \vdash a : \star \qquad \frac{\mathbb{T} \vdash \sigma_1 : \star \quad \cdots \quad \mathbb{T} \vdash \sigma_k : \star}{\mathbb{T} \vdash T\sigma_1 \ldots \sigma_k : \star}, \quad \mathit{arity}(T) = k$$

Note that the star ($\star$) does not play the same role as the star ($*$) in Barendregt's λ-cube. The intuition behind the $\star$ is that all types that may be used as terms are inhabitants of $\star$. Thus saying: "This type is well-formed" is equivalent to "This type may be used as a term".

Type variables may always be used as terms, type pattern variables only when they are introduced by the surrounding expression, and composite types only when each of their components may be used.

Definition 7. *Two types are* equivalent *in $\mathbb{E}$ (notation $\sigma \sim \tau$) iff $\mathbb{E} \vdash \sigma \sim \tau$ can be derived using the following rules:*

$$\mathbb{E}, \sigma \sim \tau \vdash \sigma \sim \tau \qquad \frac{\mathbb{E} \vdash \sigma \sim \tau}{\mathbb{E} \vdash \tau \sim \sigma} \qquad \frac{\mathbb{E} \vdash \sigma \sim \rho \qquad \mathbb{E} \vdash \rho \sim \tau}{\mathbb{E} \vdash \sigma \sim \tau}$$

$$\frac{\mathbb{E} \vdash \mathsf{T}\ \vec{\sigma} \sim \mathsf{T}\ \vec{\tau}}{\mathbb{E} \vdash \sigma_i \sim \tau_i} \qquad \frac{\mathbb{E} \vdash \sigma \sim \tau}{\mathbb{E} \vdash \rho[\alpha := \sigma] \sim \rho[\alpha := \tau]}$$

Typing Rules for the Constructs **Dynamic construction:** Values of type $\mathsf{Dynamic}$ can be constructed by pairing an expression and a type. Constraints on this pairing are that the expression is typable with the type $(E : \sigma)$ and that the type is well-formed $(\sigma : \star)$:

$$\frac{\mathbb{T} \ \vdash \ \sigma : \star \qquad \mathbb{B}, \mathbb{T}, \mathbb{E} \ \vdash \ E : \sigma}{\mathbb{B}, \mathbb{T}, \mathbb{E} \ \vdash \ \langle \sigma, E \rangle : D}, \quad \mathsf{TV}(\sigma) \cap \mathsf{TV}(\mathbb{B}) = \emptyset \quad (\mathsf{Dynamic}\text{ construction})$$

where D represents the type of dynamic expressions ($\mathsf{Dynamic}$).

Note that the constraint is only about variables in $\mathbb{B}$. Due to the introduction of type dependent functions, this constraint is not needed for type pattern variables in $\mathbb{T}$.

open construct: A *dynamic* can unpaired using the open-construct, where the value component is bound to a term variable and the type component to a type pattern variable.

When typing the expression (E_2) one may assume that the term variable is typable with the type pattern variable $(x : a)$, which is represented by adding this to $\mathbb{B}$, and that a may be used (restrictively) as a term in *Expr*, which is represented by adding $a : \star$ to $\mathbb{T}$.

As with existentially quantified types, neither x nor a may be used outside the scope of the open-expression. In particular, is it not allowed that the type of an open-expression depends on a.

This amounts to the following type rule for open-expressions ($\mathsf{Dynamic}$ destruction):

$$\frac{\mathbb{B}, \mathbb{T}, \mathbb{E} \ \vdash \ E_1 : D \qquad \mathbb{B}, x{:}a, \ \mathbb{T}, a{:}\star \ \vdash \ E_2 : \tau}{\mathbb{B}, \mathbb{T}, \mathbb{E} \ \vdash \ \underline{\mathsf{open}} \ E_1 \ \underline{\mathsf{as}} \ \langle a, x \rangle \ \underline{\mathsf{in}} \ E_2 : \tau}, \quad a \notin \mathsf{TV}(\tau)$$

typecase construct: The $\mathsf{typecase}$-construct enables matching on typeconstructors. It requires that the type a that is matched upon is well-formed $(a : \star)$. And in the branch E_i it may be assumed that the type pattern variables $\vec{a_i}$ are well-formed $(\vec{a_i}{:}\vec{\star})$.

When typing the branch E_i, the selector type-expression a may be freely substituted for the type pattern $T_i \ \vec{a_i}$ and vice versa, which is reflected by adding $a \sim T_i \ \vec{a_i}$ to the equivalence basis $\mathbb{E}$.

An auxiliary type rule states that if two types are equivalent, the one may be substituted for the other:

$$\frac{\mathbb{B}, \mathbb{T}, \mathbb{E} \ \vdash \ E : \sigma \qquad \mathbb{E} \ \vdash \ \sigma \sim \tau}{\mathbb{B}, \mathbb{T}, \mathbb{E} \ \vdash \ E : \tau} \quad \text{(substitution rule)}$$

Now the typing rule for the $\mathsf{typecase}$-construct can be formulated $(\forall_{1 \le i \le n})$:

$$\frac{\mathbb{T} \ \vdash \ a : \star \quad \mathbb{B}, \ \mathbb{T}, \vec{a_i}{:}\vec{\star}, \ \mathbb{E}, a \sim T_i \ \vec{a_i} \vdash \ E_i : \tau \quad \mathbb{B}, \mathbb{T}, \mathbb{E} \ \vdash \ E_{fail} : \tau}{\mathbb{B}, \mathbb{T}, \mathbb{E} \ \vdash \left(\begin{array}{ll} \underline{\mathsf{typecase}} \ a \ \underline{\mathsf{of}} \ \ldots & \\ \qquad\qquad T_i \ \vec{a_i} & = E_i \ \ldots \\ \qquad\qquad \underline{\mathsf{fail}} & = E_{fail} \end{array} \right) : \tau}, \quad \vec{a_i} \notin \mathsf{FV}(\tau)$$

As with the open-construct, the variables $\vec{a_i}$, which are bound by the typecase may not occur in the result type τ.

equal construct: Types can be compared (even unified) using the equal construct. It is required that both types that are to be compared are well-formed ($a : \star, \quad b : \star$). Furthermore when typing E_{succ} the types a and b can be used interchangeable.

This results in the following type rule for an equal-expression (type equality):

$$\frac{\mathbb{T} \vdash a : \star \quad \mathbb{T} \vdash b : \star \quad \mathbb{B}, \mathbb{T}, \mathbb{E}, a \sim b \vdash E_{succ} : \rho \quad \mathbb{B}, \mathbb{T}, \mathbb{E} \vdash E_{fail} : \rho}{\mathbb{B}, \mathbb{T}, \mathbb{E} \vdash \begin{array}{l}\underline{\mathsf{equal}}\ a\ b\ \underline{\mathsf{success}} = E_{succ} \\ \qquad\quad\ \underline{\mathsf{fail}} \quad\ = E_{fail}\end{array} : \rho}$$

coerce construct: The type rule for the coerce construct is analogous to that of the equal construct (type coercion):

$$\frac{\mathbb{T} \vdash a : \star \quad \mathbb{T} \vdash b : \star \quad \mathbb{B}, \mathbb{T}, \mathbb{E}, a \sim b \vdash E_{succ} : \rho \quad \mathbb{B}, \mathbb{T}, \mathbb{E} \vdash E_{fail} : \rho}{\mathbb{B}, \mathbb{T}, \mathbb{E} \vdash \begin{array}{l}\underline{\mathsf{coerce}}\ a\ b\ \underline{\mathsf{success}} = E_{succ} \\ \qquad\quad\ \underline{\mathsf{fail}} \quad\ = E_{fail}\end{array} : \rho}$$

function definition: A function definition $\mathbf{F} \quad \vec{a} \ \vec{x} = \ E$ with standard type $\Pi \ \vec{a} \ . \ \vec{\sigma} \triangleright \tau$ is called *type correct* if

$$\vec{a}{:}\vec{\star}, \ \vec{x}{:}\vec{\sigma} \vdash \ E : \tau$$

Note that the type variables $\vec{a}$ may occur in $\vec{\sigma}$ as well as in τ.

function application:

$$\frac{\mathcal{F} \vdash \mathbf{F} : \Pi \ \vec{a} \ . \ \vec{\sigma} \triangleright \tau \qquad \mathbb{T} \vdash \vec{\rho}{:}\vec{\star} \qquad \mathbb{B}, \mathbb{T}, \mathbb{E} \vdash \vec{E}{:}\vec{\sigma} \ [\vec{a}{:=}\vec{\rho}]}{\mathbb{B}, \mathbb{T}, \mathbb{E} \vdash \mathbf{F}(\rho_1, \ldots, \rho_j, E_1, \ldots, E_k) : \tau[\vec{a}{:=}\vec{\rho}]}$$

where $\mathbf{F}$ has arity k and type arity j.

In appendix A the type rules of all syntactical constructs are summarised, including those that are defined in Barendsen and Smetsers [3].

3.4 Decidability of Type Checking

This section shows the decidability of type checking by presenting an algorithm.

Definition 8. *(i) A type equation $\sigma \sim \tau$ is* distributed *over a type ρ (notation $\rho^{[\sigma \sim \tau]}$) using the following inductive definition:*

$$\begin{aligned}
\rho^{[a \sim \sigma]} &= \rho[a := \sigma] \\
\rho^{[\sigma \sim a]} &= \rho^{[a \sim \sigma]} \\
\rho^{[\mathsf{T}\vec{\sigma} \sim \mathsf{T}\vec{\tau}]} &= \rho^{[\sigma_1 \sim \tau_1] \cdots [\sigma_k \sim \tau_k]} \\
\rho^{[\sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_2]} &= \rho^{[\sigma_1 \sim \tau_1][\sigma_2 \sim \tau_2]} \\
\rho^{[\sigma \sim \tau]} &= \rho \quad , \quad \text{for all other type equations } \sigma \sim \tau
\end{aligned}$$

(ii) The distribution extends naturally to bases and expressions that contain types.

Let E be the expression that is to be typed with σ, $\mathbb{B} = \{x : \alpha, \ldots\}$ a term basis in which all free term variables in the expression have been assigned a fresh type, and $Tenv = \{a : \star \ldots\}$ a type basis that contains all free type pattern variables, then we look for a type checking algorithm that complies with the following specification:

$$\mathsf{check}(\mathbb{B}, \mathbb{T}, E, \sigma) = \begin{cases} * & \text{, if } * \text{ is the most general substitution such that} \\ & \quad \mathbb{B}^*, \mathbb{T} \vdash E^* : \sigma^* \\ \mathsf{fail} & \text{, if } E \text{ is not typable} \end{cases}$$

The implementation is given in appendix B, where the following auxiliary functions have been used:

- A unify function. This function,applied two types σ and τ, yields a most general substitution $*$ such that $\sigma^* = \tau^*$ if the types are unifiable and yields fail if they are not (see Robinson [9]).
- The operator $\oplus$ concatenates two substitutions and applies the second substitution to the arguments of the first. (Note that although this is intuitively clear, it is not a formal specification.)

3.5 Operational Semantics for the New Constructs

This section presents the evaluation rules of the new constructs of the formal language. They are given in the same notation as was used in Abadi et al. [1].

Definition 9. *An expression E is in* canonical form *if either*

$$\begin{array}{ll} E \equiv \mathbf{C}\ (E_1, \ldots, E_k) & \textit{and } E_1, \ldots, E_k \textit{ are in canonical form, or} \\ E \equiv \mathbf{F}\ E_1 \ldots E_i & \textit{and } i < \mathrm{arity}_{\mathbf{F}} \textit{ and } E_1, \ldots, E_i \textit{ are in canonical form, or} \\ E \equiv \langle \sigma, E \rangle & \textit{and } \sigma \textit{ and } E \textit{ are in canonical form, or} \\ E \equiv \mathsf{wrong} & \end{array}$$

A type σ is in canonical form *if either*

$$\begin{array}{ll} \sigma \equiv \alpha & \textit{, or} \\ \sigma \equiv \mathsf{T}\ \sigma_1 \ldots \sigma_j & \textit{and } \sigma_1, \ldots, \sigma_j \textit{ are in canonical form, or} \\ \sigma \equiv \sigma_1 \to \sigma_2 & \textit{and } \sigma_1 \textit{ and } \sigma_2 \textit{ are in canonical form, or} \\ \sigma \equiv \mathsf{wrong} & \end{array}$$

where wrong is the value of erroneous computations such as the application of one integer to another. This value is different from $\perp$ (nondetermination), which allows us to distinguish between programs that loop forever and programs that crash.

Below, we define the judgments *"expression E reduces to canonical expression* v*"* and *"type σ reduces to canonical type* t*"*, written as "$\mathsf{E} \Longrightarrow \mathsf{v}$" and "$\sigma \Longrightarrow \mathsf{t}$", for each new syntactical construct of the formal language. These judgments are presented in the same way as the rules for type assignment.

The evaluation of a **dynamic construct** involves the evaluation of both the expression and the type. If either the type or the expression reduces to wrong, then the whole dynamic reduces to wrong.

$$\frac{\vdash\ \sigma \implies \mathsf{t} \qquad\qquad \vdash\ E \implies \mathsf{v}}{\vdash\ \langle \sigma, E\rangle \implies \langle \mathsf{t}, \mathsf{v}\rangle}$$

$$\frac{\vdash\ \sigma \implies \mathsf{wrong}}{\vdash\ \langle \sigma, E\rangle \implies \mathsf{wrong}} \qquad \frac{\vdash\ E \implies \mathsf{wrong}}{\vdash\ \langle \sigma, E\rangle \implies \mathsf{wrong}}$$

The evaluation of an **open construct** consists of the evaluation of its dynamic expression, the binding of the resulting canonical forms to the variables a and x, and the evaluation of the body expression, in which a and x have been substituted for their canonical forms. If the dynamic expression does not reduce to a canonical dynamic, or the body expression reduces to wrong, then the whole open expression reduces to wrong.

$$\frac{\vdash\ E \implies \langle \mathsf{t}, \mathsf{v}\rangle \qquad\qquad \vdash\ E'[a := \mathsf{t}, x := \mathsf{v}] \implies \mathsf{v}'}{\vdash\ \underline{\mathsf{open}}\ E\ \underline{\mathsf{as}}\ \langle a, x\rangle\ \underline{\mathsf{in}}\ E' \implies \mathsf{v}'}$$

$$\frac{\vdash\ E \implies t' \quad (\ t' \text{ not of the form } \langle \mathsf{t}, \mathsf{v}\rangle\ )}{\vdash\ \underline{\mathsf{open}}\ E\ \underline{\mathsf{as}}\ \langle a, x\rangle\ \underline{\mathsf{in}}\ E' \implies \mathsf{wrong}}$$

For the evaluation of a **typecase construct**, an auxiliary function is needed that unifies two canonical types. If unification is successful, this it yields a substitution $*$ for all type (pattern) variables that occur in both canonical types, and otherwise fail is returned.

A typecase expression is evaluated by evaluating its selector type and matching this against all patterns. The result of the evaluation is the canonical form of the expression that goes with the first pattern for which the unification with the evaluated selector expression succeeds. If no unification succeeds, then the fail expression is evaluated.

$$\frac{\vdash\ a \implies \mathsf{t} \quad \forall j < i.\mathsf{unify}(\mathsf{t},\ \mathsf{T}_j\ \vec{a_j}) = \mathsf{fail} \quad \mathsf{unify}(\mathsf{t},\ \mathsf{T}_i\ \vec{a_i}) = * \quad \vdash\ E_i^* \implies \mathsf{v}}{\vdash\ \begin{array}{l} \underline{\mathsf{typecase}}\ a\ \underline{\mathsf{of}}\ \ldots \mathsf{T}_i\ \vec{a_i} = E_i\ \ldots \\ \qquad\qquad\quad \underline{\mathsf{fail}}\ \ = E_{fail} \end{array} \implies \mathsf{v}}$$

$$\frac{\vdash\ a \implies \mathsf{t} \qquad \forall i\ .\ \mathsf{unify}(\mathsf{t},\ \mathsf{T}_i\ \vec{a_i}) = \mathsf{fail} \qquad \vdash\ E_{fail} \implies \mathsf{v}}{\vdash\ \begin{array}{l} \underline{\mathsf{typecase}}\ a\ \underline{\mathsf{of}}\ \ldots \mathsf{T}_i\ \vec{a_i} = E_i\ \ldots \\ \qquad\qquad\quad \underline{\mathsf{fail}}\ \ = E_{fail} \end{array} \implies \mathsf{v}}$$

$$\frac{\vdash\ a \implies \mathsf{wrong}}{\vdash \begin{array}{l}\underline{\mathsf{typecase}}\ a\ \underline{\mathsf{of}}\ \ldots\ \mathsf{T}_i\ \vec{a_i} = E_i\ \ldots \\ \qquad\qquad\qquad \underline{\mathsf{fail}} \quad = E_{fail}\end{array} \implies \mathsf{wrong}}$$

An **equal expression** is evaluated by evaluating both argument types, which are then unified using the unify function. If successful, the expression reduces to the success branch (to which the substitution has been applied), otherwise the result will be the canonical form of the fail branch. If either of the argument types reduces to wrong, the whole expression yields wrong.

$$\frac{\vdash\ a \implies \mathsf{t} \qquad \vdash\ b \implies \mathsf{t}' \qquad \mathsf{unify}(\mathsf{t},\ \mathsf{t}') = * \qquad \vdash\ E^*_{succ} \implies \mathsf{v}}{\vdash \begin{array}{l}\underline{\mathsf{equal}}\ a\ b\ \underline{\mathsf{success}} = E_{succ} \\ \qquad\quad \underline{\mathsf{fail}} \qquad = E_{fail}\end{array} \implies \mathsf{v}}$$

$$\frac{\vdash\ a \implies \mathsf{t} \qquad \vdash\ b \implies \mathsf{t}' \qquad \mathsf{unify}(\mathsf{t},\ \mathsf{t}') = \mathsf{fail} \qquad \vdash\ E_{fail} \implies \mathsf{v}}{\vdash \begin{array}{l}\underline{\mathsf{equal}}\ a\ b\ \underline{\mathsf{success}} = E_{succ} \\ \qquad\quad \underline{\mathsf{fail}} \qquad = E_{fail}\end{array} \implies \mathsf{v}}$$

$$\frac{\vdash\ a \implies \mathsf{wrong}}{\vdash \begin{array}{l}\underline{\mathsf{equal}}\ a\ b\ \underline{\mathsf{success}} = E_{succ} \\ \qquad\quad \underline{\mathsf{fail}} \qquad = E_{fail}\end{array} \implies \mathsf{wrong}}$$

$$\frac{\vdash\ b \implies \mathsf{wrong}}{\vdash \begin{array}{l}\underline{\mathsf{equal}}\ a\ b\ \underline{\mathsf{success}} = E_{succ} \\ \qquad\quad \underline{\mathsf{fail}} \qquad = E_{fail}\end{array} \implies \mathsf{wrong}}$$

The evaluation of a **coerce construct** differs from that of an equal construct only in the auxiliary function. Instead of unifying both argument types, it must be checked whether the first type can be coerced to the second, i.e. whether the second type is an instance of the first.

On successful coercion the success branch will be evaluated with substitutions applied to it. Otherwise the fail branch will be reduced.

$$\frac{\vdash\ a \implies \mathsf{t} \qquad \vdash\ b \implies \mathsf{t}' \qquad \mathsf{coerce}(\mathsf{t},\ \mathsf{t}') = * \qquad \vdash\ E^*_{succ} \implies \mathsf{v}}{\vdash \begin{array}{l}\underline{\mathsf{coerce}}\ a\ b\ \underline{\mathsf{success}} = E_{succ} \\ \qquad\quad \underline{\mathsf{fail}} \qquad = E_{fail}\end{array} \implies \mathsf{v}}$$

$$\frac{\vdash\ a \implies \mathsf{t} \qquad \vdash\ b \implies \mathsf{t}' \qquad \mathsf{coerce}(\mathsf{t},\ \mathsf{t}') = \mathsf{fail} \qquad \vdash\ E_{fail} \implies \mathsf{v}}{\vdash \begin{array}{l}\underline{\mathsf{coerce}}\ a\ b\ \underline{\mathsf{success}} = E_{succ} \\ \qquad\quad \underline{\mathsf{fail}} \qquad = E_{fail}\end{array} \implies \mathsf{v}}$$

$$\frac{\vdash\ a \implies \mathsf{wrong}}{\vdash\ \begin{array}{l}\underline{\mathsf{coerce}}\ a\ b\ \underline{\mathsf{success}} = E_{succ}\\ \qquad\qquad\ \underline{\mathsf{fail}} \quad\ = E_{fail}\end{array} \implies \mathsf{wrong}}$$

$$\frac{\vdash\ b \implies \mathsf{wrong}}{\vdash\ \begin{array}{l}\underline{\mathsf{coerce}}\ a\ b\ \underline{\mathsf{success}} = E_{succ}\\ \qquad\qquad\ \underline{\mathsf{fail}} \quad\ = E_{fail}\end{array} \implies \mathsf{wrong}}$$

The operational semantics for **function application** have been chosen to be call-by-value.

$$\frac{\vdash \vec{\sigma} \implies \vec{\mathsf{t}} \qquad \vdash \vec{E} \implies \vec{\mathsf{v}} \qquad \vdash\ body_{\mathbf{F}}[\vec{a}{:=}\vec{\mathsf{t}}, \vec{x}{:=}\vec{\mathsf{v}}] \implies \mathsf{v}'}{\vdash\ \mathbf{F}(\sigma_1, \ldots, \sigma_j, E_1, \ldots, E_k) \implies \mathsf{v}'}$$

4 Related Work and Conclusions

Abadi et al. present in [1] a theoretical framework for Dynamic types, complete with an extensive review of the history of dynamic typing in statically typed languages. They also mention a large number of languages in which restricted use has been made of the dynamic concept. In [2] this theoretical framework is extended to cover polymorphism. Higher order pattern variables are introduced to be able to match against polymorphic types. We decided to deal with polymorphism differently and use ordinary type variables to accomplish this.

The latter approach has also been taken by Leroy and Mauny. They describe in [7] how a simple version of Dynamic is implemented in ML. They also specify a more complicated system of dynamics with universally and existentially quantified variables. This system uses the existentially qualified types to specify dynamic matches with incomplete type information. These are basically the same as our type pattern variables.

Neither of both systems allow abstraction over the types of dynamics beyond the scope of a function, though practice has shown that programmers using dynamics have a strong need for such a powerful tool. The type dependent functions that are described in this paper make such abstraction possible.

In Persistent Haskell [4] a similar, but more restricted form of dynamic typing and type dependent functions has been used. Two Haskell operations provide acces to persistent values:

```
get :: Persistent a ⇒ PSHandle → PSid → IO a
use :: Persistent a ⇒ PSHandle → PSid → a → IO a
```

The class Persistent is comparable with the class TC that is used in CLEAN. The PSid corresponds to a filename (Name) and the PSHandle and IO are comparable to the input and ouput of of the uniquely attributed Files. Using these similarities, the get and use operations can be implemented in CLEAN by using dynamics and type dependent functions.

```
get :: *Files → Name → (Error, a, *Files) | TC a
get files name = Read name files
where
    Read :: Name → *Files → (Error, a, *Files) | TC a
    Read filename files
        # (error, dyn, files) = ReadDynamic filename files
        = case dyn of
                (value :: a^) = (error, value, files)
                else          = (WrongType, bottom, files)
```

The difference between CLEAN and Persistent Haskell is that in the latter language the typecheck is implicit and has to be done on the moment of access to the persistent store, whereas in CLEAN the check be postponed.

The full system of dynamic types with type dependent functions will be implemented in CLEAN version 2.0.

5 Acknowledgements

I would like to thank Erik Barendsen, Sjaak Smetsers and Rinus Plasmeijer for their valuable remarks and discussions.

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.
2. M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic Typing in Polymorphic Languages. *Journal of Functional Programming*, 5(1):111–130, 1995.
3. E. Barendsen and J. Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.
4. A.J.T. Davie, K. Hammond, and J.-J. Quintela. *Efficient Persistent Haskell*, In *Draft Proc. 10th International Workshop on the Implementation of Functional Languages (IFL '98), London, England*, pages 183–194, 1998.
5. M.C.J.D. van Eekelen and M.J. Plasmeijer. *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley, 1993.
6. M.J.C.D. van Eekelen and M.J. Plasmeijer. *Concurrent Clean 1.0 Language Report,* Computing Science Institute, University of Nijmegen, `http://www.cs.kun.nl/~clean/Clean.Cleanbook.html`.
7. X. Leroy and M. Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

8. M. Pil. First Class File I/O. In *Proc. 8th. International Workshop on the Implementation of Functional Languages (IFL '96)*, Bad Godesberg, Germany, September 1996, volume 1268 of *LNCS*, pages 233–246. Springer-Verlag, 1997.
9. J. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association For Computing Machinery* 12:23–41, 1965.

A Type Rules Summarised

$$\mathbb{B}, x:\sigma \vdash x:\sigma$$

$$\frac{\mathcal{A} \vdash \mathbf{C} : \vec{\sigma} \triangleright \tau \qquad \mathbb{B},\mathbb{T},\mathbb{E} \vdash \vec{E}:\vec{\sigma}}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash \mathbf{C}(E_1,\ldots,E_k) : \tau}$$

$$\frac{\mathbb{B},\mathbb{T},\mathbb{E} \vdash E:\sigma \qquad \mathbb{B}, x:\sigma,\ \mathbb{T},\mathbb{E} \vdash E':\tau}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash \textsf{let}\ x = E\ \textsf{in}\ E' : \tau}$$

$$\frac{\mathbb{B}, \vec{x}:\vec{\sigma},\ \mathbb{T},\mathbb{E} \vdash E_i:\sigma_i \qquad \mathbb{B}, \vec{x}:\vec{\sigma},\ \mathbb{T},\mathbb{E} \vdash E':\tau}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash \textsf{letrec}\ \ \vec{x}=\vec{E}\ \ \textsf{in}\ E' : \tau}$$

$$\frac{\mathbb{B},\mathbb{T},\mathbb{E} \vdash E:\tau \quad \mathcal{A} \vdash \mathbf{C}_i : \vec{\sigma}_i \triangleright \tau_i \quad \mathbb{B}, \vec{x}_i:\vec{\sigma}_i,\ \mathbb{T},\mathbb{E} \vdash E_i:\rho}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash \textsf{case}\ E\ \textsf{of}\ \ \vec{P}=\vec{E}:\rho}\ \text{, if } P_i = \mathbf{C}_i\ \vec{x}_i$$

$$\frac{\mathbb{T} \vdash \sigma:\star \qquad \mathbb{B},\mathbb{T},\mathbb{E} \vdash E:\sigma}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash \langle\sigma, E\rangle : D}$$

$$\frac{\mathbb{B},\mathbb{T},\mathbb{E} \vdash E_1:D \qquad \mathbb{B}, x:a,\ \mathbb{T}, a:\star \vdash E_2:\tau}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash \underline{\textsf{open}}\ E_1\ \underline{\textsf{as}}\ \langle a, x\rangle\ \underline{\textsf{in}}\ E_2 : \tau}\ ,\quad a \notin \textsf{TV}(\tau)$$

$$\frac{\mathbb{B},\mathbb{T},\mathbb{E} \vdash E:\sigma \qquad \mathbb{E} \vdash \sigma \sim \tau}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash E:\tau}$$

$$\frac{\mathbb{T} \vdash a:\star \quad \mathbb{B},\ \mathbb{T}, \vec{a_i}:\vec{\star},\ \mathbb{E}, a\sim T_i\ \vec{a_i} \vdash E_i:\tau \quad \mathbb{B},\mathbb{T},\mathbb{E} \vdash E_{fail}:\tau}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash \begin{array}{l}\underline{\textsf{typecase}}\ a\ \underline{\textsf{of}} \ldots \\ \qquad T_i\ \vec{a_i} = E_i\ \ \ldots : \tau \\ \qquad \underline{\textsf{fail}}\ \ \ = E_{fail}\end{array}}\ ,\quad \vec{a_i} \notin \textsf{FV}(\tau)$$

$$\frac{\mathbb{T} \vdash a:\star \quad \mathbb{T} \vdash b:\star \quad \mathbb{B},\ \mathbb{T},\ \mathbb{E}, a\sim b \vdash E_{succ}:\rho \quad \mathbb{B},\mathbb{T},\mathbb{E} \vdash E_{fail}:\rho}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash \begin{array}{l}\underline{\textsf{equal}}\ a\ b\ \underline{\textsf{success}} = E_{succ} \\ \qquad\quad \underline{\textsf{fail}} \qquad = E_{fail}\end{array} : \rho}$$

$$\frac{\mathbb{T} \vdash a:\star \quad \mathbb{T} \vdash b:\star \quad \mathbb{B},\ \mathbb{T},\ \mathbb{E}, a\sim b \vdash E_{succ}:\rho \quad \mathbb{B},\mathbb{T},\mathbb{E} \vdash E_{fail}:\rho}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash \begin{array}{l}\underline{\textsf{coerce}}\ a\ b\ \underline{\textsf{success}} = E_{succ} \\ \qquad\quad \underline{\textsf{fail}} \qquad = E_{fail}\end{array} : \rho}$$

$$\frac{\mathcal{F} \vdash \mathbf{F} : \mathit{\Pi}\ \vec{a}\ .\ \vec{\sigma} \triangleright \tau \qquad \mathbb{T} \vdash \vec{\rho}:\vec{\star} \qquad \mathbb{B},\mathbb{T},\mathbb{E} \vdash \vec{E}:\vec{\sigma}\ [\vec{a}:=\vec{\rho}]}{\mathbb{B},\mathbb{T},\mathbb{E} \vdash \mathbf{F}(\rho_1,\ldots,\rho_j,E_1,\ldots,E_k) : \tau[\vec{a}:=\vec{\rho}]}$$

A function definition $\mathbf{F}\ \ \vec{a}\ \vec{x}= E$ with standard type $\mathit{\Pi}\ \vec{a}\ .\ \vec{\sigma} \triangleright \tau$ is called *type correct* if

$$\vec{a}:\vec{\star},\ \vec{x}:\vec{\sigma} \vdash E:\tau$$

B Type Checking Algorithm

(see also Section 3.4)

$$\begin{array}{rl}
\mathsf{check}(\mathbb{B},\mathbb{T},x,\sigma) = & \mathsf{unify}(\mathbb{B}(x),\sigma) \\
\mathsf{check}(\mathbb{B},\mathbb{T},\mathbf{C}\ \vec{E},\rho) = & \mathsf{check}(\mathbb{B},\mathbb{T},E_i,\sigma_i) \oplus \\
 & \mathsf{unify}(\rho,\tau) \\
 & \text{where } \vec{\sigma} \rhd \tau \text{ is the standard } \mathcal{A}\text{-type of } \mathbf{C} \\
\mathsf{check}(\mathbb{B},\mathbb{T},\mathsf{let}\ x = E\ \mathsf{in}\ E',\sigma) = & \mathsf{check}(\mathbb{B}\cup x{:}\alpha,\mathbb{T},E',\sigma) \oplus \\
 & \mathsf{check}(\mathbb{B},\mathbb{T},E,\alpha) \\
 & \alpha \text{ fresh} \\
\mathsf{check}(\mathbb{B},\mathbb{T},\mathsf{letrec}\ \vec{x}{=}\vec{E}\ \mathsf{in} E',\sigma) = & \bigoplus_i \mathsf{check}(\mathbb{B}\cup\ \vec{x}\vec{\alpha},\mathbb{T},E_i,\alpha_i) \oplus \\
 & \mathsf{check}(\mathbb{B}\cup\ \vec{x}\vec{\alpha},\mathbb{T},E',\sigma) \\
 & \vec{\alpha} \text{ fresh} \\
\mathsf{check}(\mathbb{B},\mathbb{T},\mathsf{case}\ E\ \mathsf{of}\ \ \vec{P}{=}\vec{E},\rho) = & \bigoplus_i(\mathsf{check}(\mathbb{B}\cup\ \vec{x}_i{:}\vec{\sigma}_i,\mathbb{T},E_i,\rho) \oplus \mathsf{unify}(\tau_i,\alpha)) \\
 & \oplus\ \ \mathsf{check}(\mathbb{B},\mathbb{T},E,\alpha)) \\
 & \text{where } P_i = \mathbf{C}_i\ \vec{x}_i \text{ and the standard } \mathcal{A}\text{-type} \\
 & \text{of } \mathbf{C}_i \text{ is } \vec{\sigma}_i \rhd \tau_i \\
 & \alpha \text{ fresh} \\
\mathsf{check}(\mathbb{B},\mathbb{T},\langle\sigma,E\rangle,\tau) = & \mathbf{if}\ \mathrm{TPV}(\sigma)\in\mathbb{T}\ \mathbf{then} \\
 & \quad \mathsf{check}(\mathbb{B},\mathbb{T},E,\sigma) \oplus \\
 & \quad \mathsf{unify}(\tau,D) \\
 & \mathbf{else}\mathsf{fail} \\
\mathsf{check}(\mathbb{B},\mathbb{T},\ \underline{\mathsf{open}}\ E\ \underline{\mathsf{as}}\ \langle a,x\rangle,\sigma) = & \mathsf{check}(\mathbb{B},\mathbb{T},E,D) \oplus \\
\underline{\mathsf{in}}\ E' \qquad\qquad & \\
 & \mathsf{check}(\mathbb{B}\cup x{:}a,\mathbb{T}\cup a{:}\star,E',\sigma) \\
\mathsf{check}(\mathbb{B},\mathbb{T},\ \underline{\mathsf{typecase}}\ a\ \underline{\mathsf{of}}\ldots,\sigma) = & \mathbf{if}\ a\in\mathbb{T}\ \mathbf{then} \\
T_i\ \vec{b}_i = E_i\ \ \ldots \qquad & \bigoplus_i\ (\mathsf{check}(\ \mathbb{B}^{[a\sim T_i\ \vec{b}_i]}, \\
\underline{\mathsf{fail}}\ \ \ = E_{fail} \qquad & \\
 & \qquad\qquad \mathbb{T}\cup\ \vec{b}_i{:}\star, \\
 & \qquad\qquad E_i^{[a\sim T_i\ \vec{b}_i]},\sigma^{[a\sim T_i\ \vec{b}_i]}) \\
 & \ \oplus\ \ \mathsf{check}(\mathbb{B},\mathbb{T},E_{fail},\sigma) \\
 & \mathbf{else}\ \mathsf{fail} \\
\mathsf{check}(\mathbb{B},\mathbb{T},\ \underline{\mathsf{equal}}\ a\ b \qquad\ ,\sigma) = & \mathbf{if}\ a\in\mathbb{T}\wedge b\in\mathbb{T}\ \mathbf{then} \\
\underline{\mathsf{success}} = E_{succ} \qquad & \quad \mathsf{check}(\mathbb{B}^{[a\sim b]},\mathbb{T},E_{succ}^{[a\sim b]},\sigma^{[a\sim b]}) \oplus \\
\underline{\mathsf{fail}}\ \ \ \ \ \ = E_{fail} \qquad & \quad \mathsf{check}(\mathbb{B},\mathbb{T},E_{fail},\sigma) \\
 & \mathbf{else}\ \mathsf{fail} \\
\mathsf{check}(\mathbb{B},\mathbb{T},\ \underline{\mathsf{coerce}}\ a\ b \qquad\ ,\sigma) = & \mathbf{if}\ a\in\mathbb{T}\wedge b\in\mathbb{T}\ \mathbf{then} \\
\underline{\mathsf{success}} = E_{succ} \qquad & \quad \mathsf{check}(\mathbb{B}^{[a\sim b]},\mathbb{T},E_{succ}^{[a\sim b]},\sigma^{[a\sim b]}) \oplus \\
\underline{\mathsf{fail}}\ \ \ \ \ \ = E_{fail} \qquad & \quad \mathsf{check}(\mathbb{B},\mathbb{T},E_{fail},\sigma) \\
 & \mathbf{else}\ \mathsf{fail}
\end{array}$$