D.F.R. van Arkel

Annotated Types

A System for Constrained Datatypes in Functional Languages

Leiden April 1998

Contents

We examine the use of user-defined type annotations combined with an extended type system to efficiently enforce datatype constraints. Examples of the type of constraint we have in mind are lists which are sorted and trees which are balanced. The aim is to provide language support for enforcing these constraints such that only a small number of functions need to be manually verified, general functions can be applied to the constrained datatype, multiple simultaneous constraints can be easily handled and pattern matching is possible with constrained datatypes.

We first examine the motivation for this extension of the type system. Section two gives a short summary of classical typing. Section three introduces annotated types. Section four describes unification in the presence of annotations. Section five gives a typing algorithm for annotated types along the lines of Milner's algorithm. It also gives some theoretical results for this algorithm. Next we extend this algorithm from expressions to rewrite rules and then to a complete functional language. Finally we give some examples and then conclude.

1 Motivation

This thesis examines the use of user defined annotations combined with a classical Hindley-Milner [Hindley69, Milner78] style type system to efficiently enforce datatype constraints. Examples of the type of constraints we have in mind are lists which are sorted and trees which are balanced.

We would like language support for enforcing constraints on datatypes subject to the following requirements:

- only require manual verification of constraints for a small number of functions
- allow application of general (unconstrained) functions to the constrained datatype, i.e. map is defined for plain lists, we would like to also be able to apply map to sorted lists
- we want to allow multiple constraints in parallel, i.e. sorted and non-empty lists
- we want to be able to use pattern matching with constrained datatypes

Using the plain type does not enforce the constraint thus requiring manual constraint checking of all functions where the constrained datatype occurs.

Using an ADT guarantees constraint satisfaction when the programmer shows the correctness of the signature and it does this without runtime costs but that only satisfies the first requirement. Special wrapper functions (which do add runtime cost) need to be written for each general function that we want to use with the constrained datatype, multiple constraints require new ADT's for each combination of constraints and pattern matching is not possible [BurtCam93].

We want to maintain the good qualities of ADT's and add our remaining requirements. User defined annotations provide this [Koopman96a, Koopman96b, Koopman96c]. They form a simple subtype system giving an effective solution.

We use curly brackets to indicate annotations, i.e.

$$\textbf{take} :: \textbf{Num}\ \{S\}[t] \to \{S\}[t]$$

would be the signature for a take function operating on sorted lists (if we assume S to be the annotation for sortedness and $[t]$ to be the constructor for lists of t).

But we want more. The same take function if applied to a plain list would yield a plain list as result

$$\textbf{take} :: \textbf{Num}\ [t] \to [t].$$

To express this form of type polymorphism we introduce annotation variables allowing us to state the signature of take as

$$\textbf{take} :: \textbf{Num}\ \{s : S\}[t] \to \{s : S\}[t]$$

concisely expressing the characteristics of the take function.

2 Conventional Typing

This presentation of conventional typing is taken from [BarSmets95].

We investigate typing for the following formal language:

$$E ::= x \mid \mathbf{F}(E_1, \ldots, E_k) \mid \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \mid \mu x[E]$$

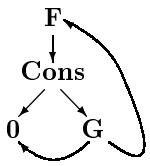
where x ranges over term variables and $\mathbf{F}$ over a set of symbols with fixed arity. We will use the notation $\vec{E}$ as shorthand for $(E_1, \ldots, E_k)$.

Ordinary sharing can be expressed using a let construct and direct cyclic dependencies with μ.

Thus the expression

$$\mathsf{let}\ x = \mathbf{0}\ \mathsf{in}\ \mu z[\mathbf{F}(\mathbf{Cons}(x, \mathbf{G}(x, z)))]$$

denotes the graph



Given a set of type variables V and a set of type constructors C then types T are recursively defined by:

$$\sigma ::= \alpha \mid \sigma_1 \to \sigma_2 \mid \mathbf{C}\vec{\sigma}$$

where α ranges over type variables V, $\mathbf{C}$ over type constructors C, $\to$ is the standard type constructor for functions and $\vec{\sigma}$ stands for $(\sigma_1, \ldots, \sigma_k)$. We will use $\sigma, \sigma_1, \ldots, \sigma_i, \tau, \tau_1, \ldots, \tau_i$ to indicate arbitrary types.

Type Environment $\mathcal{E}$ supplies types for symbols, that is it contains declarations

$$\mathbf{F} : \vec{\sigma} \rightarrowtail \tau$$

This includes both function symbols and data constructor symbols. We use $\vec{\sigma} \rightarrowtail \tau$ as shorthand for $(\sigma_1, \sigma_2, \ldots, \sigma_k) \to \tau$.

We use the notation $\mathcal{E}(\mathbf{f}) = \vec{\sigma} \rightarrowtail \tau$ if $\mathbf{f} : \vec{\sigma} \rightarrowtail \tau$ in $\mathcal{E}$.

Basis B: a finite set of variable declarations of the form $x : \tau$.

We use the notation $B(x) = \tau$ if $x : \tau$ in B and $B[x : \tau] = [x_1 : \tau_1, \ldots, x_k : \tau_k, x : \tau]$ if $B = [x_1 : \tau_1, \ldots, x_k : \tau_k]$.

A substitution θ is a mapping from type variables to types, written as $[\alpha_1 : \tau_1, \ldots]$, $[\,]$ is the empty substitution. $\theta(\alpha) = \tau$ if $\alpha : \tau$ in θ. Uniform substitution, written as $\theta\tau$, is defined as follows:

$$\begin{array}{lll} \theta\alpha & = \tau & \text{if } \theta(\alpha) = \tau \\ & = \alpha & \text{otherwise} \\ \theta(\sigma \to \tau) & = \theta\sigma \to \theta\tau & \\ \theta\mathbf{C}(\sigma_1, \ldots, \sigma_k) & = \mathbf{C}(\theta\sigma_1, \ldots, \theta\sigma_k) & \end{array}$$

We define substitution composition $\theta\theta'$ as $\theta\theta'\sigma = \theta(\theta'\sigma)$.

We say that θ is idempotent if $\theta\theta = \theta$ which is to say that if $\theta\sigma = \tau$, then $\theta\tau = \tau$.

This system deals with statements of the form

$$B \vdash_{\mathcal{E}} E : \sigma$$

with the interpretation that given a basis B we can derive that expression E has type σ using type environment $\mathcal{E}$. Such a statement is valid if it can be produced by the following derivation rules:

$$\frac{}{B(x) = \sigma \vdash_{\mathcal{E}} x : \sigma} \text{(VARIABLE)}$$

$$\frac{B \vdash_{\mathcal{E}} \vec{E} : \vec{\sigma} \quad \mathcal{E} \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau}{B \vdash_{\mathcal{E}} \mathbf{F}\vec{E} : \tau} \text{(APPLICATION)}$$

$$\frac{B \vdash_{\mathcal{E}} E_1 : \sigma_1 \quad B[x : \sigma_1] \vdash_{\mathcal{E}} E_2 : \sigma_2}{B \vdash_{\mathcal{E}} \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \sigma_2} \text{(SHARING)}$$

$$\frac{B[x : \sigma] \vdash_{\mathcal{E}} E : \sigma}{B \vdash_{\mathcal{E}} \mu x[E] : \sigma} \text{(CYCLE)}$$

Due to the separation of the specifications of rewrite rules and algebraic types from their applications we need a mechanism to deal with different occurences of symbols. This is called instantiation and is handled by the next two rules:

$$\frac{}{\mathcal{E}(\mathbf{F}) = \vec{\sigma} \rightarrowtail \tau \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau} \text{(INSTANTIATION)}$$

$$\frac{\mathcal{E} \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau}{\mathcal{E} \vdash \mathbf{F} : \theta\vec{\sigma} \rightarrowtail \theta\tau} \text{(SUBSTITUTION)}$$

where θ is an idempotent substitution.

A term is called algebraic if it is built up from data constructors and variables only, using application.

$\mathcal{FV}(E)$ denotes the set of free variables occuring in expression E:

$$\begin{array}{l}\mathcal{FV}(x) = \{x\} \\ \mathcal{FV}(\mathbf{F}(E_1, \ldots, E_k)) = \bigcup_{i=1}^{k} \mathcal{FV}(E_i) \\ \mathcal{FV}(\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2) = \mathcal{FV}(E_1) \cup \mathcal{FV}(E_2) \setminus x \\ \mathcal{FV}(\mu x[E]) = \mathcal{FV}(E) \setminus x\end{array}$$

Operations on terms are defined using rewrite rules of the form

$$\mathbf{F}(A_1, \ldots, A_k) \rightarrow E,$$

where $\mathcal{FV}(E) \subseteq \mathcal{FV}(\vec{A})$, and the A_i are algebraic expressions. Following standard practice in functional languages such as Clean [PlasEek97] only left-linear rewrite rules are considered i.e. rules in which each variable occurs at most once in the left-hand side. For example

$$\mathbf{Append}(\mathbf{Cons}(h, t), l) \rightarrow \mathbf{Cons}(h, \mathbf{Append}(t, l)).$$

The rewrite semantics of expressions (according to some set $\mathcal{R}$ of rules) is defined as usual. By $E \underset{\mathcal{R}}{\twoheadrightarrow} E'$ we denote that E rewrites to E' in zero or more steps.

We say that $\mathcal{E}$ is $\mathcal{R}$-OK if for each rule

$$\mathbf{F}(A_1, \ldots, A_k) \to E,$$

say with $\mathcal{E}$ containing $\mathbf{F} : (\sigma_1, \ldots, \sigma_k) \rightarrowtail \tau$ one has for some B

$$B \vdash_{\mathcal{E}} A_1 : \sigma_1, \ldots, B \vdash_{\mathcal{E}} A_k : \sigma_k, B \vdash_{\mathcal{E}} E : \tau.$$

For example if $\mathcal{E}$ contains

Append : $([s],[s]) \rightarrowtail [s]$
Cons : $(s,[s]) \rightarrowtail [s]$

and for B

$B(h) = s$
$B(t) = [s]$
$B(l) = [s]$

then the example rule above is $\mathcal{R}$-OK.

The following two theorems give the two main properties for classical typing. Proofs can be found in [Barendsen95].

The first states that typing is preserved during reduction.

Theorem 1 (subject reduction property).

$$\left.\begin{array}{r} B \vdash_{\mathcal{E}} E : \sigma \\ E \underset{\mathcal{R}}{\twoheadrightarrow} E' \\ \mathcal{E} \text{ is } \mathcal{R}\text{-OK} \end{array}\right\} \Rightarrow B \vdash_{\mathcal{E}} E' : \sigma$$

The second states that each typeable expression E has a type from which all other types for E can be obtained by instantiation.

Theorem 2 (principal type property). *Suppose E is typeable. Then there exist B_0, σ_0 such that for any B and σ*

$$B \vdash_{\mathcal{E}} E : \sigma \Rightarrow B \supseteq \theta B_0, \sigma = \theta\sigma_0$$

for some substitution θ.

3 Annotated Typing

In order to distinguish the rules for annotated typing from those for ordinary typing we use $\triangleright$ instead of $\vdash$ and prime the names of the rules.

Given a set of type variables V, a set of type constructors C and a set of type annotations $\mathcal{A}$ then the annotated types T_A are recursively defined by:

$$\sigma ::= \alpha \mid \sigma_1 \to \sigma_2 \mid \langle \mathbf{C}, \mathcal{A} \rangle \vec{\sigma}$$

where α ranges over type variables V and $\mathbf{C}$ over type constructors C. $\mathcal{A} = \{a : A, \dots\}$ where a ranges over $\mathbf{0}$, $\mathbf{1}$ and annotation variables. A ranges over annotation symbols.

We use the annotation symbol to indicate the type of constraint, for example S for sorted and N for non-empty lists. $\mathbf{0}$ indicates that the constraint is not (known to be) present, $\mathbf{1}$ indicates that the constraint is present and annotation variables are used to indicate type polymorphism. If we use $\{A\}[\alpha]$ as notation for $\langle \mathbf{List}, A \rangle \alpha$ then

$$\mathbf{Tail} : \{\mathbf{1} : N, s : S\}[\alpha] \rightarrowtail \{s : S\}[\alpha]$$

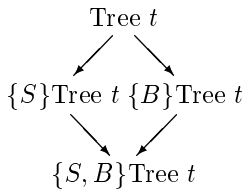
indicates that **Tail** is a function that takes a list of α and produces a list of α, and the argument list should be non-empty $\{\mathbf{1} : N\}$ and furthermore that if the argument is sorted then the result is sorted. Thus **Tail** preserves the sortedness constraint if it is present.

We assume that if for a particular annotation A no annotation is given, then $\mathbf{0} : A$ is implied.

We define the operation $|\sigma|$ to find the underlying (unannotated) type as follows:

$$\begin{array}{ll} |\alpha| & = \alpha \\ |\sigma_1 \to \sigma_2| & = |\sigma_1| \to |\sigma_2| \\ |\langle \mathbf{C}, \mathcal{A} \rangle (\sigma_1, \dots, \sigma_k)| & = \mathbf{C}(|\sigma_1|, \dots, |\sigma_k|) \end{array}$$

Annotations induce a subtype relation on types. For example if trees can have annotations S and B for sorted and balanced trees respectively then we get the following partial order:



The subtype ($\leq$) relation on annotated types:

$$\frac{}{\sigma \leq \sigma} \text{(ID)}$$

$$\frac{\vec{\sigma} \leq \vec{\sigma}' \quad \mathcal{A}' \subseteq \mathcal{A}}{\langle \mathbf{C}, \mathcal{A} \rangle \vec{\sigma} \leq \langle \mathbf{C}, \mathcal{A}' \rangle \vec{\sigma}'} \text{(SUBTYPE)}$$

$$\frac{\sigma_1 \leq \sigma_1' \quad \dots \quad \sigma_k \leq \sigma_k'}{\vec{\sigma} \leq \vec{\sigma}'} \text{(VECTOR)}$$

$$\frac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'} \text{ (CONTRAVARIANT)}$$

$\subseteq$ on annotation sets is defined component-wise where $\subseteq$ on annotations:

$$\begin{array}{lcl} \mathbf{0} : A & \subseteq & \mathbf{1} : A \\ \mathbf{0} : A & \subseteq & v : A \\ v : A & \subseteq & \mathbf{1} : A \\ \mathbf{0} : A & \subseteq & \mathbf{0} : A \\ \mathbf{1} : A & \subseteq & \mathbf{1} : A \\ v : A & \subseteq & v : A \end{array}$$

With annotated types the variable, application and sharing rules remain the same. The cycle rule on the other hand needs to be modified giving:

$$\frac{B[x : \sigma] \rhd_{\mathcal{E}} E : \tau \quad \tau \leq \sigma}{B \rhd_{\mathcal{E}} \mu x[E] : \tau} \text{ (CYCLE}')$$

We only need this modification for technical reasons in our typing algorithm which follows later. Note that if we restrict this to underlying types it is equivalent to the original rule.

A substitution now consists of two parts. One being the underlying substitution which is the classical substitution given before but now extended to be a mapping from type variables to annotated types. The other being the annotation substitution which is a mapping from annotation variables to annotations. We use the notation $|\theta|$ for the underlying substitution and θ_A for the annotation substitution parts of θ. Uniform substitution is now defined as follows:

$$\begin{array}{lll} \theta\alpha & = \tau & \text{if } |\theta|(\alpha) = \tau \\ & = \alpha & \text{otherwise} \\ \theta(\sigma \to \tau) & = \theta\sigma \to \theta\tau & \\ \theta\langle \mathbf{C}, \mathcal{A} \rangle(\sigma_1, \ldots, \sigma_k) & = \langle \mathbf{C}, \mathcal{A}' \rangle(\theta\sigma_1, \ldots, \theta\sigma_k) & \end{array}$$

where $\mathcal{A}' = \{a' : A \mid a : A \in \mathcal{A} \wedge a \mapsto a' \in \theta_A\} \cup \{a : A \mid a : A \in \mathcal{A} \wedge a \notin \mathcal{D}om(\theta_A)\}$. This corresponds with replacing annotation variables given in the mapping with their replacements and leaving all other annotations intact.

We define $\mathcal{D}om(\theta_A) = \{a \mid a \mapsto a' \in \theta_A\}$

The new form of substitution means that although the SUBSTITUTION rule still looks exactly the same:

$$\frac{\mathcal{E} \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau}{\mathcal{E} \vdash \mathbf{F} : \theta\vec{\sigma} \rightarrowtail \theta\tau} \text{ (SUBSTITUTION}')$$

it now has a broader interpretation.

For example given

$$\mathbf{F} : (\{a : A\}\mathbf{C}(\ldots), \{a : A\}\mathbf{C}(\ldots)) \rightarrowtail \{a : A\}\mathbf{C}(\ldots)$$

then one can always use

$$\mathbf{F} : (\{\mathbf{0} : A\}\mathbf{C}(\ldots), \{\mathbf{0} : A\}\mathbf{C}(\ldots)) \rightarrowtail \{\mathbf{0} : A\}\mathbf{C}(\ldots)$$

and

$$\mathbf{F} : (\{\mathbf{1} : A\}\mathbf{C}(\ldots), \{\mathbf{1} : A\}\mathbf{C}(\ldots)) \rightarrowtail \{\mathbf{1} : A\}\mathbf{C}(\ldots)$$

Besides the modified interpretation of the substitution rule we also need an extra instantiation rule:

$$\frac{\mathcal{E} \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau \quad \sigma_i' \leq \sigma_i}{\mathcal{E} \vdash \mathbf{F} : \vec{\sigma}' \rightarrowtail \tau} \text{(WEAKENING}')$$

This allows weakening of the argument type required for function applications. Intuitively it reflects the fact that one may always reduce the type of accepted arguments for a function to an instance of the most general type allowed i.e. for the example above this means that we may also use

$$\mathbf{F} : (\{\mathbf{0} : A\}\mathbf{C}(\dots), \{\mathbf{1} : A\}\mathbf{C}(\dots)) \rightarrowtail \{\mathbf{0} : A\}\mathbf{C}(\dots)$$

and

$$\mathbf{F} : (\{\mathbf{1} : A\}\mathbf{C}(\dots), \{\mathbf{0} : A\}\mathbf{C}(\dots)) \rightarrowtail \{\mathbf{0} : A\}\mathbf{C}(\dots)$$

We do not need a similar rule for the result type since that wil occur as an argument elsewhere and can be weakened if necessary then.

Summary of the derivation rules for annotated types:

$$\frac{}{B(x) = \sigma \rhd_{\mathcal{E}} x : \sigma} \text{(VARIABLE}')$$

$$\frac{B \rhd_{\mathcal{E}} \vec{E} : \vec{\sigma} \quad \mathcal{E} \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau}{B \rhd_{\mathcal{E}} \mathbf{F}\vec{E} : \tau} \text{(APPLICATION}')$$

$$\frac{B \rhd_{\mathcal{E}} E_1 : \sigma_1 \quad B[x : \sigma_1] \rhd_{\mathcal{E}} E_2 : \sigma_2}{B \rhd_{\mathcal{E}} \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \sigma_2} \text{(SHARING}')$$

$$\frac{B[x : \sigma] \rhd_{\mathcal{E}} E : \tau \quad \tau \leq \sigma}{B \rhd_{\mathcal{E}} \mu x[E] : \tau} \text{(CYCLE}')$$

Summary of the instantiation rules for annotated types:

$$\frac{}{\mathcal{E}(\mathbf{F}) = \vec{\sigma} \rightarrowtail \tau \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau} \text{(INSTANTIATION}')$$

$$\frac{\mathcal{E} \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau}{\mathcal{E} \vdash \mathbf{F} : \theta\vec{\sigma} \rightarrowtail \theta\tau} \text{(SUBSTITUTION}')$$

where θ is an idempotent substitution.

$$\frac{\mathcal{E} \vdash \mathbf{F} : \vec{\sigma} \rightarrowtail \tau \quad \sigma_i' \leq \sigma_i}{\mathcal{E} \vdash \mathbf{F} : \vec{\sigma}' \rightarrowtail \tau} \text{(WEAKENING}')$$

4 Unification of Annotated Types

The next step is to define unification in the presence of annotations. This is done in such a way that for the underlying types it is equivalent to Robinson unification $\mathcal{U}$ [Robinson65, Robinson71].

Unfortunately we do not retain all the properties of Robinson unification, whereas Robinson unification is symmetric in its two arguments annotated unification cannot be. For example succesful unification of a sorted list with a plain list depends on which one is the expected type and which one is the actual type.

$$\left.\begin{array}{l}\mathcal{E}(\mathbf{f}) = ([\sigma]) \rightarrowtail \tau \\ B(v) = \{\mathbf{1} : S\}[\sigma]\end{array}\right\} \Longrightarrow \mathbf{f}(v) : \tau$$

whereas

$$\left.\begin{array}{l}\mathcal{E}(\mathbf{f}) = (\{\mathbf{1} : S\}[\sigma]) \rightarrowtail \tau \\ B(v) = [\sigma]\end{array}\right\} \Longrightarrow \text{ Fail, no type can be derived for } \mathbf{f}(v).$$

Therefore we will define unification such that it differentiates between its two arguments, taking the first to be the expected argument and the second to be the actual argument. Furthermore, if unification succeeds then the actual argument wil be a subtype of the expected argument.

4.1 Unification of annotated type constructors

We start by examining unification of two annotated type constructors. Given

$$\mathcal{E}(\mathbf{f}) = (\{s : S\}[\sigma]) \rightarrowtail \{s : S\}[\sigma]$$

then we want

$$\begin{array}{ll}B(v) = \{\mathbf{1} : S\}[\sigma] & \Longrightarrow \mathbf{f}(v) : \{\mathbf{1} : S\}[\sigma] \\ B(v) = \{\mathbf{0} : S\}[\sigma] & \Longrightarrow \mathbf{f}(v) : \{\mathbf{0} : S\}[\sigma] \\ B(v) = \{t : S\}[\sigma] & \Longrightarrow \mathbf{f}(v) : \{t : S\}[\sigma]\end{array}$$

This seems to indicate that we can always substitute the actual annotation for the annotation variable in the expected argument but it is not quite that simple. Given

$$\mathcal{E}(\mathbf{f}) = (\{s : S\}[\sigma], \{s : S\}[\sigma]) \rightarrowtail \{s : S\}[\sigma]$$

and

$$B(v) = \{\mathbf{1} : S\}[\sigma]$$

then we want

$$\begin{array}{ll}B(w) = \{\mathbf{1} : S\}[\sigma] & \Longrightarrow \mathbf{f}(v, w) : \{\mathbf{1} : S\}[\sigma] \\ B(w) = \{\mathbf{0} : S\}[\sigma] & \Longrightarrow \mathbf{f}(v, w) : \{\mathbf{0} : S\}[\sigma] \\ B(w) = \{t : S\}[\sigma] & \Longrightarrow \mathbf{f}(v, w) : \{t : S\}[\sigma]\end{array}$$

but if

$$B(v) = \{\mathbf{0} : S\}[\sigma]$$

then we want

$$\begin{array}{ll}B(w) = \{\mathbf{1} : S\}[\sigma] & \Longrightarrow \mathbf{f}(v, w) : \{\mathbf{0} : S\}[\sigma] \\ B(w) = \{\mathbf{0} : S\}[\sigma] & \Longrightarrow \mathbf{f}(v, w) : \{\mathbf{0} : S\}[\sigma] \\ B(w) = \{t : S\}[\sigma] & \Longrightarrow \mathbf{f}(v, w) : \{\mathbf{0} : S\}[\sigma]\end{array}$$

and if

$$B(v) = \{t : S\}[\sigma]$$

then we want

$$\begin{array}{ll} B(w) = \{\mathbf{1} : S\}[\sigma] & \Longrightarrow \mathbf{f}(v, w) : \{t : S\}[\sigma] \\ B(w) = \{\mathbf{0} : S\}[\sigma] & \Longrightarrow \mathbf{f}(v, w) : \{\mathbf{0} : S\}[\sigma] \\ B(w) = \{u : S\}[\sigma] & \Longrightarrow \mathbf{f}(v, w) : \{u : S\}[\sigma] \end{array}$$

where in the final case we require that u and t are identified with each other.

Thus if the expected argument has annotation $\mathbf{0}$ we can ignore the actual annotation. If the expected argument has an annotation $\mathbf{1}$ then if the actual annotation is $\mathbf{0}$ unification fails, if it is $\mathbf{1}$ unification succeeds and if it is a variable v then unification succeeds with $v \mapsto \mathbf{1}$ in the resulting substitution. If the expected argument has annotation v then if the actual annotation is $\mathbf{0}$ then unification succeeds with $v \mapsto \mathbf{0}$ in the resulting substitution. If the actual annotation is a variable w then unification succeeds with $v \mapsto w$ in the resulting substitution and if the actual annotation is $\mathbf{1}$ then unification succeeds without any addition to the resulting substitution.

*expected**actual*	$\mathbf{0}$	β	$\mathbf{1}$
$\mathbf{0}$	✓	✓	✓
α	✓$(\alpha \mapsto \mathbf{0})$	✓$(\alpha \mapsto \beta)$	✓
$\mathbf{1}$	×	✓$(\beta \mapsto \mathbf{1})$	✓

4.2 Unification of a type variable with a type constructor

The next issue arises when unifying a type variable with an annotated type constructor. We first look at the case when the type variable occurs in the expected argument.

$$\begin{array}{l} \mathcal{E}(\mathbf{f}) = (\alpha) \rightarrowtail \alpha \\ B(v) = \langle \mathbf{C}, \mathcal{A} \rangle \sigma \end{array}$$

Initially we would expect that $\mathbf{f}(v) : \langle \mathbf{C}, \mathcal{A} \rangle \sigma$, but things are not that simple. To demonstrate this we extend the example:

$$\begin{array}{l} \mathcal{E}(\mathbf{f}) = (\alpha, \alpha) \rightarrowtail \alpha \\ B(v) = \langle \mathbf{C}, \mathcal{A}_1 \rangle \sigma_1 \\ B(w) = \langle \mathbf{C}, \mathcal{A}_2 \rangle \sigma_2 \end{array}$$

Now we would expect that $\mathbf{f}(v) : \langle \mathbf{C}, \mathcal{A}_1 \rangle \sigma_1 \rightarrowtail \langle \mathbf{C}, \mathcal{A}_1 \rangle \sigma_1$ and to proceed with unification from there. When we give concrete types for v and w the problem shows itself

$$\begin{array}{l} B(v) = \{\mathbf{1} : S\}[\mathbf{Num}] \\ B(w) = \{\mathbf{0} : S\}[\mathbf{Num}] \end{array}$$

then $\mathbf{f}(v) : \{\mathbf{1} : S\}[\mathbf{Num}] \rightarrowtail \{\mathbf{1} : S\}[\mathbf{Num}]$ and unification for $\mathbf{f}(v, w)$ fails.

This leads us to propose the following rule for the single argument case: $\mathbf{f}(v) : \langle \mathbf{C}, \mathcal{A}' \rangle \sigma'$ provided that $\alpha \notin \mathcal{V}ar(\sigma)$, otherwise unification fails, and $\langle \mathbf{C}, \mathcal{A} \rangle \sigma \leq \langle \mathbf{C}, \mathcal{A}' \rangle \sigma'$.

Here $\mathcal{V}ar$ is a utility function giving the set of type variables in a type as follows:

$$\begin{array}{l} \mathcal{V}ar(\alpha) = \alpha \\ \mathcal{V}ar(\sigma \rightarrow \tau) = \mathcal{V}ar(\sigma) \cup \mathcal{V}ar(\tau) \\ \mathcal{V}ar(\langle \mathbf{C}, \mathcal{A} \rangle \vec{\sigma}) = \mathcal{V}ar(\vec{\sigma}) \\ \mathcal{V}ar((\sigma_1, \ldots, \sigma_k)) = \bigcup_{i=1}^{k} \mathcal{V}ar(\sigma_i) \end{array}$$

For the two argument case we propose: $\mathbf{f}(v,w) : \langle \mathbf{C}, \mathcal{A}' \rangle \sigma'$ provided that $\alpha \notin \mathcal{V}ar(\sigma_1)$ and $\alpha \notin \mathcal{V}ar(\sigma_2)$ with $\langle \mathbf{C}, \mathcal{A}_1 \rangle \sigma_1 \leq \langle \mathbf{C}, \mathcal{A}' \rangle \sigma'$ and $\langle \mathbf{C}, \mathcal{A}_2 \rangle \sigma_2 \leq \langle \mathbf{C}, \mathcal{A}' \rangle \sigma'$.

This leads to the rule that if an annotation has value $\mathbf{0}$ in $\mathcal{A}_1$ then it has value $\mathbf{0}$ in an initial substitution for α. Similarly a value of v is retained but a value of $\mathbf{1}$ is modeled by a fresh annotation variable w in the initial substitution for α. Then in the next step this initial substitution for α is unified with $\langle \mathbf{C}, \mathcal{A}_2 \rangle \sigma_2$ as described earlier.

4.3 Unification of a type constructor with a type variable

Next we look at the case when the type variable occurs in the actual argument.

$\mathcal{E}(\mathbf{f}) = (\langle \mathbf{C}, \mathcal{A}_1 \rangle \sigma_1, \langle \mathbf{C}, \mathcal{A}_2 \rangle \sigma_2) \rightarrowtail \tau$
$B(v) = \alpha$
$B(w) = \alpha$

then $\mathbf{f}(v,w) : \tau$ and $\alpha : \langle \mathbf{C}, \mathcal{A}' \rangle \sigma'$ where $\langle \mathbf{C}, \mathcal{A}' \rangle \sigma' \leq \langle \mathbf{C}, \mathcal{A}_1 \rangle \sigma_1$ and $\langle \mathbf{C}, \mathcal{A}' \rangle \sigma' \leq \langle \mathbf{C}, \mathcal{A}_2 \rangle \sigma_2$.

This leads to the rule that if an annotation has value $\mathbf{0}$ in $\mathcal{A}_1$ then it is modeled by a fresh annotation variable w in an initial substitution for α. Otherwise if an annotation has a value of v or $\mathbf{1}$ then that value is retained in the initial substitution for α. In the next step this initial substitution for α is unified with $\langle \mathbf{C}, \mathcal{A}_2 \rangle \sigma_2$ as described earlier.

Further examination reveals that both these cases are equivalent to initially assuming a substitution for α where each annotation has a fresh annotation variable as value.

4.4 Complete algorithm

We now present the complete unification algorithm with first the expected argument and then the actual argument reflecting *unify*(exp,act). The algorithm returns the substitution required to unify its two arguments and if this is not possible it returns FAIL.

$unify(\alpha, \alpha)$	$= [\,]$
$unify(\alpha, \beta)$	$= [\beta : \alpha]$
$unify(\alpha, \sigma_1 \rightarrow \sigma_2)$	$= [\alpha : \theta\alpha_1 \rightarrow \theta\alpha_2]$ if $\alpha \notin \mathcal{V}ar(\sigma_1 \rightarrow \sigma_2)$
	where α_1, α_2 fresh and $\theta = unify((\alpha_1, \alpha_2), (\sigma_1, \sigma_2))$
	= FAIL otherwise
$unify(\alpha, \langle \mathbf{C}, \mathcal{A} \rangle \vec{\sigma})$	$= unify(\langle \mathbf{C}, \mathcal{A}' \rangle \vec{\sigma}', \langle \mathbf{C}, \mathcal{A} \rangle \vec{\sigma})$ if $\alpha \notin \mathcal{V}ar(\vec{\sigma})$
	where $\vec{\sigma}'$ fresh and $A' = copy(A)$
	= FAIL otherwise
$unify(\sigma_1 \rightarrow \sigma_2, \alpha)$	$= [\alpha : \theta\alpha_1 \rightarrow \theta\alpha_2]$ if $\alpha \notin \mathcal{V}ar(\sigma_1 \rightarrow \sigma_2)$
	where α_1, α_2 fresh and $\theta = unify((\sigma_1, \sigma_2), (\alpha_1, \alpha_2))$
	= FAIL otherwise
$unify(\sigma_1 \rightarrow \sigma_2, \sigma_1' \rightarrow \sigma_2')$	$= unify((\sigma_1, \sigma_2), (\sigma_1', \sigma_2'))$
$unify(\sigma_1 \rightarrow \sigma_2, \langle \mathbf{C}, \mathcal{A} \rangle \vec{\sigma})$	= FAIL
$unify(\langle \mathbf{C}, \mathcal{A} \rangle \vec{\sigma}, \alpha)$	$= unify(\langle \mathbf{C}, \mathcal{A} \rangle \vec{\sigma}, \langle \mathbf{C}, \mathcal{A}' \rangle \vec{\sigma}')$ if $\alpha \notin \mathcal{V}ar(\vec{\sigma})$
	where $\vec{\sigma}'$ fresh and $A' = copy(A)$
	= FAIL otherwise
$unify(\langle \mathbf{C}, \mathcal{A} \rangle \vec{\sigma}, \sigma_1 \rightarrow \sigma_2)$	= FAIL

$$
\begin{array}{ll}
\mathit{unify}(\langle \mathbf{C}, \mathcal{A}\rangle\vec{\sigma}, \langle \mathbf{C}, \mathcal{A}'\rangle\vec{\sigma}') & = \theta'\theta \\
 & \quad \text{where } \theta = \mathit{anunify}(A, A') \\
 & \quad \text{and } \theta' = \mathit{unify}(\theta\vec{\sigma}, \theta\vec{\sigma}') \\
\mathit{unify}(\langle \mathbf{C}, \mathcal{A}\rangle\vec{\sigma}, \langle \mathbf{C}', \mathcal{A}'\rangle\vec{\sigma}') & = \text{FAIL} \\
\mathit{unify}((\sigma_1, \sigma_2, \ldots, \sigma_k), (\sigma'_1, \sigma'_2, \ldots, \sigma'_k)) & = \theta'\theta \\
 & \quad \text{where } \theta = \mathit{unify}(\sigma_1, \sigma'_1) \\
 & \quad \text{and } \theta' = \mathit{unify}((\theta\sigma_2, \ldots, \theta\sigma_k), (\theta\sigma'_2, \ldots, \theta\sigma'_k))
\end{array}
$$

Anunify carries out the actual annotation unification in accordance with the tabel given on page 10 and *copy* is a function that copies the annotations but with fresh annotation variables as values.

We should really pass a third argument and return a tuple as result to make the passing of the fresh variable supply explicit. The modifications required are obvious and not given here.

4.5 Unification properties

We now proceed to the properties of our unification algorithm.

Theorem 3. *If* $\mathit{unify}(\sigma, \tau) = \theta$ *then* θ *is idempotent.*

Proof. This is a direct consequence of the fact that we avoid introducing direct circularities and in substitution compositions $\theta'\theta$ the variables which fall in the range of substitution θ are substituted away in the expressions whose unification resulted in θ' and thus cannot occur as a result of applying that substitution. □

The next theorem states that when types are restricted to their underlying unannotated variants *unify* and $\mathcal{U}$ are equivalent. The proof remains future work since that would require a complete formal presentation of $\mathcal{U}$.

Hypothesis 4.

$$
\left.\begin{array}{l} \mathit{unify}(\sigma, \tau) = \theta \\ \mathcal{U}(|\sigma|, |\tau|) = \theta' \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} |\theta\sigma| = \theta'|\sigma| \\ |\theta\tau| = \theta'|\tau| \end{array}\right.
$$

The final theorem states that *unify* satisfies our original requirement that if unification succeeds then the actual argument is a subtype of the expected argument.

Theorem 5. *If* $\mathit{unify}(\sigma, \tau) = \theta$ *then* $|\theta\sigma| = |\theta\tau|$ *and* $\theta\sigma \leq \theta\tau$

Proof. The first part follows directly from the previous theorem combined with the standard result for $\mathcal{U}$ that $\mathcal{U}(\sigma, \tau) = \theta \implies \theta\sigma = \theta\tau$. The second part directly follows from the style of annotation unification chosen. □

5 Algorithm $\mathcal{W}$

We now present the complete typing algorithm for expressions with annotated types in the style of Milner's algorithm $\mathcal{W}$. It takes a four-tuple (F, B, E, e) as argument where F is the supply of fresh type and annotation variables, B is the current basis, E is the environment and e is the expression to be typed. It then returns a three-tupel (F', θ, τ) where F' is the partially depleted fresh variable supply, θ is a substitution and τ is a type. The intended result now is that if $\mathcal{W}(F, B, E, e) = (F', \theta, \tau)$ then the judgement

$$\theta B, E \rhd e : \theta\tau$$

is valid.

$$
\begin{array}{l}
\mathcal{W} : (F, B, E, e) \rightarrowtail (F', \theta, \tau) \\
\mathcal{W}(F, B, E, e) = \textbf{case } e \textbf{ of} \\
\qquad\qquad x \quad \Rightarrow \quad \textbf{let} \\
\qquad\qquad\qquad\qquad\quad \tau = B(x) \\
\qquad\qquad\qquad\quad \textbf{in } (F, \{\}, \tau) \\
\qquad\qquad f\vec{e} \quad \Rightarrow \quad \textbf{let} \\
\qquad\qquad\qquad\qquad\quad (\vec{\sigma} \rightarrowtail \tau, F') = \mathit{instantiate}(E(f), F) \\
\qquad\qquad\qquad\qquad\quad (F'', \theta, \vec{\sigma}') = \mathcal{W}(F', B, E, \vec{e}\,) \\
\qquad\qquad\qquad\qquad\quad (F''', \theta') = \mathit{unify}(F'', \theta\vec{\sigma}, \vec{\sigma}') \\
\qquad\qquad\qquad\quad \textbf{in } (F''', \theta'\theta, \tau) \\
\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \quad \Rightarrow \quad \textbf{let} \\
\qquad\qquad\qquad\qquad\quad (F', \theta, \tau) = \mathcal{W}(F, B, E, e_1) \\
\qquad\qquad\qquad\qquad\quad B' = B[x : \tau] \\
\qquad\qquad\qquad\qquad\quad (F'', \theta', \sigma) = \mathcal{W}(F', B', E, e_2) \\
\qquad\qquad\qquad\quad \textbf{in } (F'', \theta'\theta, \sigma) \\
\qquad\qquad \mu x[e] \quad \Rightarrow \quad \textbf{let} \\
\qquad\qquad\qquad\qquad\quad (\sigma, F') = \mathit{fresh}(F) \\
\qquad\qquad\qquad\qquad\quad (F'', \theta, \tau) = \mathcal{W}(F', B[x : \sigma], E, e) \\
\qquad\qquad\qquad\qquad\quad (F''', \theta') = \mathit{unify}(F'', \theta\sigma, \theta\tau) \\
\qquad\qquad\qquad\quad \textbf{in } (F''', \theta'\theta, \tau) \\
\qquad\qquad (e, \vec{e}\,) \quad \Rightarrow \quad \textbf{let} \\
\qquad\qquad\qquad\qquad\quad (F', \theta, \tau) = \mathcal{W}(F, B, E, e) \\
\qquad\qquad\qquad\qquad\quad (F'', \theta', \vec{\tau}) = \mathcal{W}(F', B, E, \vec{e}\,) \\
\qquad\qquad\qquad\quad \textbf{in } (F'', \theta'\theta, (\tau, \vec{\tau}\,))
\end{array}
$$

Here *fresh* is a function that returns a fresh type variable from the fresh variable supply. *Instantiate* is a function that takes a type and a fresh variable supply and returns a duplicate of the type but with fresh type and annotation variables and the depleted fresh variable supply.

This algorithm is a direct implementation of the typing rules given and directly compares to its classical counterpart. All the real work involving annotations is taken care of in the unification algorithm.

We now proceed to a few formal results for our algorithm. The first lemma states that the subtype relationship is maintained under substitution.

Lemma 1.

$$\forall\theta : \sigma \leq \tau \implies \theta\sigma \leq \theta\tau$$

Proof. with induction to the structure of types. Note that $\sigma \leq \tau$ implies that $|\sigma| = |\tau|$.

$\sigma = \tau = \alpha$:
then $\theta\sigma = \theta\tau = \rho$ if $\alpha : \rho$ in θ otherwise $\theta\sigma = \theta\tau = \alpha$. In either case $\theta\sigma \leq \theta\tau$ according to the ID rule.

$\sigma = \rho \to \rho'$ and $\tau = \delta \to \delta'$:
then $\theta\sigma = \theta\rho \to \theta\rho'$ and $\theta\tau = \theta\delta \to \theta\delta'$. Since $\sigma \leq \tau$ then $\delta \leq \rho$ and $\rho' \leq \delta'$. Applying induction we find that $\theta\delta \leq \theta\rho$ and $\theta\rho' \leq \theta\delta'$. Applying the CONTRAVARIANT rule for function types we find that $\theta\rho \to \theta\rho' \leq \theta\delta \to \theta\delta'$ resulting in $\theta\sigma \leq \theta\tau$.

$\sigma = \langle \mathbf{C}, \mathcal{A}_\sigma \rangle \vec{\rho}$ and $\tau = \langle \mathbf{C}, \mathcal{A}_\tau \rangle \vec{\rho}\,'$:
then $\theta\sigma = \langle \mathbf{C}, \mathcal{A}'_\sigma \rangle \theta\vec{\rho}$ and $\theta\tau = \langle \mathbf{C}, \mathcal{A}'_\tau \rangle \theta\vec{\rho}\,'$. Now $\theta\vec{\rho} \leq \theta\vec{\rho}\,'$ according to the induction hypothesis. It remains to be shown that $\mathcal{A}_\sigma \leq \mathcal{A}_\tau \implies \mathcal{A}'_\sigma \leq \mathcal{A}'_\tau$ which a simple case analysis shows to be true. □

The second lemma states that if we apply a substitution on an entire type inference it remains valid.

Lemma 2.

$$\forall\theta : B \rhd_\mathcal{E} e : \sigma \implies \theta B \rhd_\mathcal{E} e : \theta\sigma$$

Proof. We use induction on the structure of the inference.

Basis:

For the VARIABLE′ rule this is trivial.

Induction:

If the APPLICATION′ rule was applied then we know that $B \rhd_\mathcal{E} \vec{e} : \vec{\sigma}'$, $\mathcal{E} \vdash f : \vec{\sigma} \to \tau$ and $\sigma_i \leq \sigma'_i$. Applying induction on $B \rhd_\mathcal{E} \vec{e} : \vec{\sigma}'$ allows us to conclude that $\theta B \rhd_\mathcal{E} \vec{e} : \theta\vec{\sigma}'$. With the SUBSTITUTION rule we conclude from $\mathcal{E} \vdash f : \vec{\sigma} \to \tau$ that $\mathcal{E} \vdash f : \theta\vec{\sigma} \to \theta\tau$. Now we apply lemma 1 to $\sigma_i \leq \sigma'_i$ giving $\theta\sigma_i \leq \theta\sigma'_i$. Finally we can combine these three results with APPLICATION′ to give $\theta B \rhd_\mathcal{E} f\vec{e} : \theta\tau$.

SHARING′: then we know that $B \rhd_\mathcal{E} e_1 : \sigma_1$ and $B, x : \sigma_1 \rhd_\mathcal{E} e_2 : \sigma_2$. From $B \rhd_\mathcal{E} e_1 : \sigma_1$ according to induction $\theta B \rhd_\mathcal{E} e_1 : \theta\sigma_1$. And also from $B, x : \sigma_1 \rhd_\mathcal{E} e_2 : \sigma_2$ according to induction $\theta B, x : \theta\sigma_1 \rhd_\mathcal{E} e_2 : \theta\sigma_2$. Applying SHARING′ gives $\theta B \rhd_\mathcal{E} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \theta\sigma_2$

CYCLE′: then we know $B, x : \sigma \rhd_\mathcal{E} e : \tau$ and $\sigma \leq \tau$. Applying induction on the first and the first lemma on the second item gives $\theta B, x : \theta\sigma \rhd_\mathcal{E} e : \theta\tau$ and $\theta\sigma \leq \theta\tau$. Finally with CYCLE′: $\theta B \rhd_\mathcal{E} \mu x[e] : \theta\tau$ □

With the aid of the previous two lemmas we are now able to prove the correctness of the typing algorithm with respect to the inference rules.

Theorem 6 ($\mathcal{W}$ is correct). *If* $\mathcal{W}(F, B, \mathcal{E}, e) = (F', \theta, \tau)$ *then* $\theta B, \mathcal{E} \rhd e : \theta\tau$

Proof. We use induction on the structure of the expression e.

$\mathcal{W}(F, B, \mathcal{E}, x)$: the algorithm gives $(F, \{\}, B(x))$ which can be directly emulated by a single application of the VARIABLE′ rule.

$\mathcal{W}(F, B, \mathcal{E}, f\vec{e})$: from the induction hypothesis we find that $\theta B \rhd_{\mathcal{E}} \vec{e} : \theta\vec{\sigma}'$. The second lemma allows the conclusion $\theta'\theta B \rhd \vec{e} : \theta'\theta\vec{\sigma}'$. The SUBSTITUTION rule allows $\mathcal{E} \vdash f : \theta'\theta\vec{\sigma} \rightarrowtail \theta'\theta\tau$. The first lemma allows $\theta'\theta\sigma_i \leq \theta'\theta\sigma_i'$ which together with the observation that $\theta'\theta' = \theta'$ (since θ' is idempotent) gives $\theta'\theta'\theta\sigma_i \leq \theta'\theta\sigma_i'$. Now we can use the APPLICATION$'$ rule to conclude that $\theta'\theta B \rhd_{\mathcal{E}} f\vec{e} : \theta'\theta'\theta\tau$, again using $\theta'\theta' = \theta'$ this gives the desired result that $\theta'\theta B \rhd_{\mathcal{E}} f\vec{e} : \theta'\theta\tau$

$\mathcal{W}(F, B, \mathcal{E}, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$: from the induction hypothesis we find that $B, \mathcal{E} \rhd e_1 : \tau$ and $B, x : \tau \rhd e_2 : \sigma$. Application of the SHARING$'$ rule allows us to conclude that $B, \mathcal{E} \rhd \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \sigma$ as required.

$\mathcal{W}(F, B, \mathcal{E}, \mu x[e])$: from the induction hypothesis we find that $\theta B[x : \theta\sigma] \rhd_{\mathcal{E}} e : \theta\tau$. Application of the second lemma and the CYCLE$'$ rule gives the desired result. □

Finally we would like a theorem stating that principal types exist for annotated types and that $\mathcal{W}$ derives these. For the underlying types this is clear since both the typing and the unification algorithm are equivalent to their classical counterparts for underlying types. It is as yet unclear if it is also valid for annotated types and this remains for future investigation.

Hypothesis 7 ($\mathcal{W}$ approximates principal types). *If $B, \mathcal{E} \rhd e : \tau$ and $F \cap \mathcal{FV}(B) = \varnothing$ then there is a substitution δ such that $\mathcal{W}(F, B', \mathcal{E}, e) = (F', \theta, \tau')$ with $B \supseteq \delta\theta B'$ and $\tau = \delta\theta\tau'$.*

With respect to the subject reduction property this remains valid for the underlying types but not for the annotations. For example given a sort function that sorts lists then **Sort** $[3, 2]$ with type $\{\mathbf{1} : S\}[\mathbf{Num}]$ reduces to $[2, 3]$ with type $\{\mathbf{0} : S\}[\mathbf{Num}]$. However this is no worse than the problems that occur with type synonyms and abstract data types. In an ADT the derived type after some reductions can be completely different from the original type whereas with annotations only some of the annotations can be lost. For example with the following ADT for sorted lists

```
definition module slist

:: Slist t

new :: Slist t
head :: (Slist t) -> t
tail :: (Slist t) -> Slist t
insert :: t (Slist t) -> Slist t
isEmpty :: (Slist t) -> Bool
```

and the following possible implementation

```
implementation module slist

:: Slist t :== [t]

new :: Slist t
new = []

...
```

then the expression `new` has type `Slist t` whereas its reduct `[]` will be assigned type `[t]`.

Thus we can conclude the following properties for the annotated type system:

- The principal type property remains valid for the underlying unannotated types.
- The subject reduction property remains valid for the underlying unannotated types, but not for the annotations.
- If we ignore the presence of annotations we retain the classical Hindley-Milner type system.

6 Typing rewrite rules

With this algorithm for expression typing we can develop a system to typecheck rewrite rules. Recall that these have the form

$$\mathbf{F}(A_1, \ldots, A_k) \to E$$

and that if $\mathcal{E}(\mathbf{F}) = (\sigma_1, \ldots, \sigma_k) \rightarrowtail \tau$ we require that $B \vdash_{\mathcal{E}} A_1 : \sigma_1, \ldots, B \vdash_{\mathcal{E}} A_k : \sigma_k, B \vdash_{\mathcal{E}} E : \tau$ for some B. Furthermore recall that we require that $\mathcal{FV}(E) \subseteq \mathcal{FV}(\vec{A})$ and that each free variable occurs only once in the left hand side. To type a rewrite rule we first type $\vec{A}$ with an initial basis in which the free variables receive a free type variable. Then we type the right hand side expression E with the resulting basis.

Rather than requiring equality between the type inferred for the rule and the type found in the environment we relax this to requiring that the environment type for each of the arguments is a subtype of the inferred type and that the inferred type of the result is a subtype of the environment type. This is applied in turn to each of the rule alternatives.

With respect to the environment we need to distinguish between the environment for typing the pattern and for typing the right hand side expression. A good example of why this is useful is the list constructor. In the pattern we can assign it a type of $(\alpha, \{s : S\}[\alpha]) \rightarrowtail \{s : S\}[\alpha]$ but in the right hand side expression it has type $(\alpha, [\alpha]) \rightarrowtail [\alpha]$. We can then infer that $\mathbf{Tail}(\mathbf{Cons}(h, t)) \to t$ has type $(\{s : S\}[\alpha]) \rightarrowtail \{s : S\}[\alpha]$ whereas $\mathbf{Construct}(h, t) \to \mathbf{Cons}(h, t)$ has type $(\alpha, [\alpha]) \rightarrowtail [\alpha]$.

The extension of this system with currying and higher-order functions can be done as described in Barendsen [Barendsen95]. The idea is to associate with each function symbol $\mathbf{F}$ of non-zero arity (say n) a number of so-called Curry variants of arity $0, 1, \ldots, n-1$ respectively. Next we add rules for the special function symbol $\mathbf{Ap}$ stating

$$\begin{aligned}
\mathbf{Ap}(\mathbf{F_0}, x_1) &\to \mathbf{F_1}(x_1) \\
\mathbf{Ap}(\mathbf{F_1}(x_1), x_2) &\to \mathbf{F_2}(x_1, x_2) \\
&\vdots \\
\mathbf{Ap}(\mathbf{F_{n-1}}(x_1, \ldots, x_{n-1}), x_n) &\to \mathbf{F}(x_1, \ldots, x_n)
\end{aligned}$$

Now we can simulate higher order functions as follows. Say we want to model the function $\mathbf{G}(x) = \lambda y.\mathbf{F}(x, y)$ then given a rule

$$\mathbf{F}(x, y) \to \cdots$$

then the application $\mathbf{F_1}(x)$ exactly stands for the required $\lambda y.\mathbf{F}(x, y)$

7 Typing Simple Clean

We have used this system as the basis for a typesystem for a simple subset of Clean extended with notations for annotations. This simple Clean differs from Clean in that there is no overloading, strictness, uniqueness, user-defined operators, guards, records, local function definitions or list comprehensions. For the rest they have the same syntax.

The typesystem for this language uses the system given for rewrite rules to type the various rules in the source with a few extra additions. These are:

Function types If a type for a function is given then that is taken to be the environment type. For each rule alternative the given type for each of the arguments has to be a subtype or an instance of the inferred type. The opposite applies for the result type.

If no type is given then the inferred type of the first rule alternative is used as the environment type for the other alternatives. This last approach is clearly a crude approximation which would benefit from further work.

Function type assertions Besides the usual $fie :: \sigma \to \tau$ we have added $fie :! \sigma \to \tau$ for those cases where the annotations of the result type τ cannot be inferred. For example a sort function on lists would have a sorted list as result type, this cannot be inferred by the typechecker so a function type assertion is required. The typechecker then checks the argument types as usual and restricts itself to checking the underlying type for the result.

Constructor and destructor type assertions These are for giving annotated types for the datatype constructors. Normally datatype constructors implicitly receive a type when they are introduced.

```
List t = Nil               // gives type Nil :: List t
       | Cons t (List t)   // gives type Cons :: t (List t) -> List t
```

Now we want a method of introducing annotations to the datatypes produced by constructors. For example `Cons` always produces a list that is non-empty and we might want to add an annotation for that. But as we have seen above we also want to introduce different types for applications of datatype constructors in the pattern and in the right hand side expression. We use the notation :<: to introduce types for constructor applications in patterns and :>: for constructor applications in the right hand side expression. For example

```
Nil  :>: {1:S}List t                // the empty list is always sorted
Cons :>: t (List t) -> {1:N}List t  // the constructed list is never empty
```

The advantage here is that we only need to give these constructor properties once instead of needing to explicitly specify them at every application.

With respect to the syntax we have used the usual Clean syntax with the extensions given above. Note that `where` clauses are used to introduce sharing and cycles.

8 Examples

We now present a few examples showing how type annotations can be used.

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

::Num = Zero | Succ Num

// we use the following annotations for lists :-
// N: non-empty
// S: sorted
// E: even length
// O: odd length

Nil :<: {1:S,1:E}[t]
Nil :>: {1:S,1:E}[t]


Cons :<: t {s:S,o:O,e:E}[t] -> {1:N,s:S,e:O,o:E}[t]
Cons :>: t {o:O,e:E}[t] -> {1:N,e:O,o:E}[t]
```

Note how `Cons` exchanges the annotations for odd and even length lists.

```
Head :: {1:N}[a] -> a
// we can only take the head of a non-empty list
Head (Cons h t) = h
```

The difference with the normal `Head` is that our type system now ensures that `Head` is only applied to non-empty lists. Thus there can be no runtime errors, but only programs for which the typechecker can determine that this is always the case are accepted.

```
Tail :: {1:N,s:S,o:O,e:E}[a] -> {s:S,e:O,o:E}[a]
// the tail of a list is still sorted if the whole list is sorted
Tail (Cons h t) = t

Map :: (a->b) {n:N,o:O,e:E}[a] -> {n:N,o:O,e:E}[b]
// map maintains all annotations except sortedness
Map f Nil = Nil
Map f (Cons h t) = Cons (f h) (Map f t)

Take :: Num {s:S}[t] -> {s:S}[t]
// take maintains sortedness
Take Zero _ = Nil
Take (Succ n) (Cons h t) = Cons h (Take n t)
Take _ _ = Nil
```

```
Append :: [t] [t] -> [t]
Append Nil l = l
Append (Cons h t) l = Cons h (Append t l)
```

We would like `:: {n1:N}[t] {n2:N}[t] -> {n1 v n2:N}[t]` but this is something our current system cannot handle. Similarly for the length annotations we would like an AND condition: `{o1:O,e1:E}[t] {o2:O,e2:E}[t] -> {(o1&e2)v(e1&o2):O,(e1&e2)v(o1&o2):E}[t]`

```
Insert :! a {s:S,o:O,e:E}[a] -> {1:N,s:S,e:O,o:E}[a]
// we have to assert that insertion maintains sortedness
Insert e Nil = Cons e Nil
Insert e (Cons h t) = If (e <= h) (Cons e (Cons h t)) (Cons h (Insert e t))

Sort :: {n:N,o:O,e:E}[a] -> {1:S,n:N,o:O,e:E}[a]
Sort Nil = Nil
Sort (Cons h t) = Insert h (Sort t)
```

We can give the type for sort without an assertion, the typing algorithm correctly checks this itself. Unfortunately it is not able to derive this type correctly from the rule alternatives given. This is due to the second rule alternative which is recursive. The typechecker still correctly finds the underlying type but is unable to correctly identify the annotations for odd and even length between the argument and the result.

9 Conclusion

This approach has given promising early results. We have succesfully added simple subtypes to a simple functional language through the use of annotations. This has given us language support for enforcing datatype constraints in such a manner that it satisfies all our requirements. We only need to manually check functions defined with a type assertion and the datatypes for which we have constructor and/or destructor type assertions. General functions are directly applicable to constrained datatypes. As seen in the examples multiple constraints are trivial to use requiring no more work than a single constraint and pattern matching of constrained datatypes is possible.

The nice qualities of the classical type system have been maintained over the underlying types. The principal type property is still valid over the underlying types as is subject reduction. If we ignore the presence of annotations we retain the classical Hindley-Milner type system.

More practical experience is needed to see how well this extended system combines with the full Clean language in real world programs.

For further research:

Overloading of plain functions with optimised annotated variants, i.e. the minimum of a sorted list is always the head. Here as with other forms of overloading it is the programmers responsibility to ensure that overloaded forms perform the same function. It would be a rather unfortunate situation if the overloaded form of a function produced a differing result from the plain function given the same arguments.

The interaction of annotations with typeclasses needs to be investigated. A precursory examination seems to indicate no problems. Also the interaction with other standard extensions to the type system such as strictness and uniqueness annotations should be examined.

References

[Barendsen95] E. Barendsen (1995) Types and Computations in Lambda Calculi and Graph Rewrite Systems. *Ph.D. Thesis, University of Nijmegen, The Netherlands.*

[BarSmets95] E. Barendsen and J.E.W. Smetsers (1995) A Derivation System for Uniqueness Typing. *Proc. of Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA '95), Volterra (Pisa), Italy,* 151–158. *ed. Corradini and Montaneri, Elsevier, Electronic Notes in Theoretical Computer Science.*

[BurtCam93] W.F. Burton and R.D. Cameron (1993) Pattern matching with abstract datatypes. *J. Functional Programming,* **3,** (2), 171–190.

[Hindley69] J.R. Hindley (1969) The principal type scheme of an object in combinatory logic. *Trans. American Mathematical Society,* **146,** 29–60.

[Koopman96a] P. Koopman (1996) Constrained data types. *Technical Report 96-36, Computer Science, Leiden University, The Netherlands.*

[Koopman96b] P. Koopman (1996) Language Support to Enforce Constraints on Data Types. *Technical Report 96-37, Computer Science, Leiden University, The Netherlands.*

[Koopman96c] P. Koopman (1996) Constrained data types. *in Dagstuhl Seminar Report 156, Spetember 1996.*

[Milner78] R.A. Milner (1978) Theory of type polymorphism in programming. *J. Computer and System Sciences,* **17,** (3), 348–375.

[PlasEek97] M.J. Plasmeijer and M.J.C.D. van Eekelen (1997) Concurrent Clean 1.2 Language Reference Manual. http://www.cs.kun.nl/~clean.

[Robinson65] J.A. Robinson (1965) A Machine-Oriented Logic Based on the Resolution Principle. *J. Assoc. Comput. Mach.,* **12,** 23–41.

[Robinson71] J.A. Robinson (1971) Computational Logic: The Unification Computation. *Machine Intelligence,* **6,** 63–72. *ed. B. Meltzer and D. Michie, Edinburgh University Press.*