Typechecking the Charity Term Logic

Peter Vesely

April 14, 1997

1 Introduction

The purpose of this note is to describe the term logic typechecker which has been implemented in the Charity interpreter. The typechecker plays an important role in the front end of the interpreter. In conjunction with the parser, the typechecker determines whether an expression entered by the user is valid. The typical input processing sequence is as follows:

1. The user enters an expression. For our purposes this may be a term or a function.
2. The parser determines if the expression is allowed by the rules of the grammar, that is, whether the expression is syntactically valid.
3. The typechecker determines if the expression is well-typed, that is, whether the expression is semantically valid.

An expression which is grammatically correct and well-typed can then be processed further. This may include further semantic checks (for example, checking for exhaustive patterns), translation to machine instructions, and partial or full evaluation.

2 Parse Trees

The typechecker gets its input from the parser. In this section we describe what the typechecker expects as input. We give a simplified version of the Charity grammar and also the properties that are assumed to hold of a successfully parsed expression. For our purposes, an expression passed to the typechecker can be a *term*, *function*, or *function definition*.

2.1 Patterns

The parse tree for a pattern is given by the following rules:

$$
\begin{array}{lcl}
\mathit{fo\_patt} & ::= & \mathit{term\_var} \\
 & | & \mathit{constr}\ \mathit{fo\_patt} \\
 & | & () \\
 & | & (\mathit{fo\_patt}) \\
 & | & (\mathit{fo\_patt}, \mathit{fo\_patt}) \\
 & | & (\mathit{fo\_destr} : \mathit{fo\_patt}, \mathit{ho\_destr} : \mathit{ho\_patt}, \ldots) \\
 & | & \_ \\
\mathit{ho\_patt} & ::= & \mathit{fn\_var} \\
 & | & \_
\end{array}
$$

The main role of a pattern is to introduce term variables (*term_var*'s) and function variables (*fn_var*'s) with a local scoping. Additionally, a first-order pattern (*fo_patt*) can be used to describe the domain of a function. The parse tree for a *fo_patt* is assumed to satisfy the following properties:

1. Each variable (*term_var* or *fn_var*) in a pattern may occur in that pattern no more than once.

2. In a constructor pattern, *constr fo_patt*, the *constr* must be a constructor of some previously declared inductive datatype.

3. In a record pattern, $(\mathit{fo\_destr} : \mathit{fo\_patt}, \mathit{ho\_destr} : \mathit{ho\_patt}, \ldots)$, all the destructors must belong to the same co-inductive datatype, and there must be no more than one entry for each destructor. Each destructor entry which is missing is assumed to be a don't care pattern (_).

2.2 Function Definitions

The parse tree for a function definition is given by the following rules:

$$
\begin{array}{lcl}
\mathit{defn} & ::= & \mathrm{def}\ \mathit{fn\_sym}\ =\ \mathit{function} \\[1em]
 & \mid & \mathrm{def}\ \mathit{fn\_sym}\,\{\mathit{fn\_var}\,, \ldots, \mathit{fn\_var}\}\ =\ \mathit{function}
\end{array}
$$

A function definition introduces a function symbol (*fn_sym*) which corresponds to a specified function. The function symbol can be subsequently used to stand for the fully specified function. The parse tree for a function definition satisfies the following properties:

1. The function symbol introduced by a function definition may not be used in the body of the definition (unless it is re-introduced in some pattern and used in the scope of that pattern).
2. In a function definition with function parameters, all the parameters must be distinct and they become available as function variables (*fn_var*'s) whose scope is the body of the definition.

2.3 Terms

The parse tree for a term is given by the following rules:

$$
\begin{array}{lcl}
\mathit{term} & ::= & \mathit{term\_var} \\[1em]
 & \mid & () \\[1em]
 & \mid & (\mathit{term}) \\[1em]
 & \mid & (\mathit{term}\,, \mathit{term}) \\[1em]
 & \mid & (\mathit{fo\_destr} : \mathit{term}\,, \mathit{ho\_destr} : \mathit{function}\,, \ldots) \\[1em]
 & \mid & \mathit{function}\ \mathit{term}
\end{array}
$$

The parse tree of a term is assumed to satisfy the following properties:

1. A term variable (*term_var*) must have been introduced in a pattern and occurs in the scope of that pattern.

2. In a record term, $(fo\_destr : term, ho\_destr : function, \ldots)$, all the destructors must belong to the same co-inductive datatype, and there must be exactly one entry for each destructor of the datatype. The entry for each first-order destructor is a term and the entry for each higher-order destructor is a function.

2.4 Functions

The parse tree for a function is given by the following rules:

$$
\begin{array}{rcl}
function & ::= & fn\_var \\
 & | & fn\_sym \\
 & | & fn\_sym \{ function, \ldots, function \} \\
 & | & \left\{ \begin{array}{c} fo\_patt \mapsto term \\ \cdots \\ fo\_patt \mapsto term \end{array} \right\} \\
 & | & \left\{\!\left| \begin{array}{c} constr : function \\ \cdots \\ constr : function \end{array} \right|\!\right\} \\
 & | & \left(\!\left| fo\_patt \mapsto \begin{array}{c} fo\_destr : term \\ ho\_destr : function \\ \cdots \end{array} \right|\!\right) \\
 & | & type\_con \{ map\_param, \ldots, map\_param \} \\
map\_param & ::= & function \\
 & | & function \;\&\; function \\
 & | & \_
\end{array}
$$

The parse tree for a function is assumed to satisfy the following properties:

1. A function variable (fn_var) must have been introduced in a pattern or in the function parameter list of a definition, and occurs in the scope of that pattern or parameter list.

2. A function symbol (*fn_sym*) with no parameters is either a structor of some datatype or some previously defined function with no parameters.

3. A function symbol with parameters is some previously defined function. The number of actual parameters must correspond to the number of parameters given in the definition.

4. Patterns in case functions need not be exhaustive. The typechecker only needs to ensure all patterns in a case function have the same type. Checking that the patterns are exhaustive can be performed in a later step, if typechecking succeeds.

5. A fold function must have exactly one phrase corresponding to each constructor of some previously declared inductive datatype.

6. An unfold function must have exactly one phrase corresponding to each destructor of some previously declared co-inductive datatype. The phrase for each first-order destructor must be a term and the phrase for each higher-order destructor must be a function.

7. In a map function, $\mathit{type\_con}\,\{\,\mathit{map\_param}\,, \ldots, \mathit{map\_param}\,\}$, the number of map parameters must be the same as the arity of the type constructor. A map parameter in a position of positive or negative variance is just a function. A map parameter in a position of bivariance is a pair of functions separated by the & symbol. A map parameter in a position of nonvariance is a don't care function (_).

3 Collecting Equations

The typechecker takes an expression (in the form of a parse tree satisfying the conditions given above) and tries to find its most general typing, if such a typing exists. In the case of a function definition, the expression may come with some user-specified typing information. In this case the typechecker must also determine if the user-specified typing is a valid instance of the most general typing. The role of the typechecker is then to take an expression with some typing constraints (possibly none) and to determine if the expression has a typing and, in light of any given typing constraints, what the most general typing is.

The algorithm for determining the typing of an expression involves collecting equations between types while recursively decomposing the expression. The idea is to obtain all equations which must hold for the expression to be well-typed and then to attempt to solve these equations for the unknown type variables. If a solution exists then the expression is well-typed.

In this section we describe the process of collecting equations through recursive decomposition of parse trees. Some small examples illustrating this process are shown below using the decomposition tables in 3.1, 3.2, and 3.3.

3.0.1 Example

$$\begin{array}{cl}
\vdash \{(x,y) \mapsto (y,x)\} : A_0 \to A_1 & \\ \hline
(x,y) : A_0 \vdash (y,x) : A_1 & \\ \hline
x : A_2 \, , \, y : A_3 \vdash (y,x) : A_1 & A_0 = A_2 \times A_3 \\ \hline
& A_1 = A_4 \times A_5
\end{array}$$

$$\begin{array}{c} x : A_2 \, , \, y : A_3 \vdash y : A_4 \\ \hline . \end{array} \; A_4 = A_3 \qquad \begin{array}{cl} x : A_2 \, , \, y : A_3 \vdash x : A_5 & \\ \hline x : A_2 \vdash x : A_5 & \\ \hline . & A_5 = A_2 \end{array}$$

Solving the collected equations for A_0 and A_1 gives the typing

$$\{(x,y) \mapsto (y,x)\} : A_2 \times A_3 \to A_3 \times A_2$$

Notice how we have chosen to decompose the context (left of the $\vdash$) *before* the term (right of the $\vdash$). This strategy generally results in fewer equations being generated.

$\square$

3.0.2 Example

Consider a very simple function definition in which some type information is specified:

$$\text{def } \mathit{apply}\{g : X \to Y\} \ = \ \{x \mapsto g(x)\}.$$

The decomposition proceeds as follows:

$$\dfrac{g : X \to Y \vdash \{x \mapsto g(x)\} : A_0 \to A_1}{\dfrac{g : X \to Y\,,\ x : A_0 \vdash g(x) : A_1}{\dfrac{g : X \to Y\,,\ x : A_0 \vdash x : A_2}{\cdot}\ A_2 = A_0 \qquad \dfrac{g : X \to Y\,,\ x : A_0 \vdash g : A_2 \to A_1}{\dfrac{g : X \to Y \vdash g : A_2 \to A_1}{\cdot}\ A_2 = X\,,\ A_1 = Y}}}$$

Solving the collected equations for A_0 and A_1 gives the typing

$$\mathit{apply}\{g : X \to Y\} : X \to Y$$

□

3.0.3 Example

Here is an example of an expression which fails to typecheck.

$$\dfrac{\vdash \{(x, \_) \mapsto x\}() : A_0}{\dfrac{\vdash () : A_1}{\cdot}\ A_1 = 1 \qquad \dfrac{\vdash \{(x, \_) \mapsto x\} : A_1 \to A_0}{\dfrac{\dfrac{(x, \_) : A_1 \vdash x : A_0}{x : A_2\,,\ \_ : A_3 \vdash x : A_0}\ A_1 = A_2 \times A_3}{\dfrac{x : A_2 \vdash x : A_0}{\cdot}\ A_0 = A_2}}}$$

The resulting collection of equations fails to have a solution (since 1 does not unify with $\times$).

□

3.1 Decomposing Patterns

The rules for decomposing patterns are shown below. All occurrences of A and B denote new type variables.

unit patt	$\dfrac{\Gamma\,,\,() : S \vdash \varphi}{\Gamma \vdash \varphi}\; 1 = S$
pair patt	$\dfrac{\Gamma\,,\,(p,q) : S \vdash \varphi}{\Gamma\,,\,p : A\,,\,q : B \vdash \varphi}\; A \times B = S$
record patt	$\dfrac{\Gamma\,,\,(d_i : p_i, d_j : p_j) : S \vdash \varphi}{\Gamma\,,\,p_i : F_i(A, R(A))\,,\,p_j : E_j(A) \to F_j(A, R(A)) \vdash \varphi}\; R(A) = S$
constr patt	$\dfrac{\Gamma\,,\,c_i\,p : S \vdash \varphi}{\Gamma\,,\,p : E_i(A, L(A)) \vdash \varphi}\; L(A) = S$
don't care patt	$\dfrac{\Gamma\,,\,\_ : S \vdash \varphi}{\Gamma \vdash \varphi}$

1. The *record patt* rule illustrates decomposition for both first-order destructors (d_i) and higher-order destructors (d_j). The datatype R is assumed to be of the form

$$\begin{array}{rcl} \text{data } C \to R(A) & = & \cdots \\ & \mid & d_i : C \to F_i(A, C) \\ & \mid & d_j : C \to E_j(A) \Rightarrow F_j(A, C) \\ & \mid & \cdots \end{array}$$

   Note that a record pattern is not generally required to have an entry for each destructor. Any missing destructor entries can be viewed as "don't care" patterns.

2. The *constr patt* rule illustrates decomposition of constructor patterns. The datatype L is assumed to be of the form

$$\begin{array}{rcl} \text{data } L(A) \to C & = & \cdots \\ & \mid & c_i : E_i(A, C) \to C \\ & \mid & \cdots \end{array}$$

3.2 Decomposing Terms

The rules for decomposing terms are shown below. All occurrences of A and B denote new type variables.

unit term	$\dfrac{\Gamma \vdash () : T}{\cdot}\; 1 = T$
pair term	$\dfrac{\Gamma \vdash (s,t) : T}{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}\; A \times B = T$
record term	$\dfrac{\Gamma \vdash (d_i : t_i, d_j : f_j) : T}{\Gamma \vdash t_i : F_i(A, R(A)) \qquad \Gamma \vdash f_j : E_j(A) \to F_j(A, R(A))}\; R(A) = T$
application	$\dfrac{\Gamma \vdash fs : T}{\Gamma \vdash s : A \qquad \Gamma \vdash f : A \to T}$
term variable	$\dfrac{\Gamma, x : S \vdash x : T}{\cdot}\; S = T \qquad \dfrac{\Gamma, \rho \vdash x : T}{\Gamma \vdash x : T}\; (x \notin \rho)$

1. As for record patterns, the *record term* rule illustrates term decomposition for both first-order destructors (d_i) and higher-order destructors (d_j). Unlike record patterns, however, record terms must have an entry for each destructor. In the *record term* rule the datatype R would be of the form

$$\begin{array}{rcl} \text{data } C \to R(A) & = & d_i : C \to F_i(A, C) \\ & | & d_j : C \to E_j(A) \Rightarrow F_j(A, C) \end{array}$$

2. The *pair term* rule can be viewed as a special case of the *record term* rule for the product datatype.

3. The *term variable* rules match a term variable x with its latest occurrence in the context.

3.3 Decomposing Functions

The rules for decomposing functions are shown below. All occurrences of A and B denote new type variables.

case	$$\dfrac{\Gamma \vdash \left\{\begin{array}{c} p_1 \mapsto t_1 \\ \cdots \\ p_n \mapsto t_n \end{array}\right\} : S \to T}{\Gamma\,,\, p_1 : S \vdash t_1 : T \quad \cdots \quad \Gamma\,,\, p_n : S \vdash t_n : T}$$	
fold	$$\dfrac{\Gamma \vdash \left\{\!\left| \begin{array}{c} c_1 : f_1 \\ \cdots \\ c_n : f_n \end{array} \right|\!\right\} : S \to T}{\Gamma \vdash f_1 : E_1(A,T) \to T \quad \cdots \quad \Gamma \vdash f_n : E_n(A,T) \to T}$$	$S = L(A)$
unfold	$$\dfrac{\Gamma \vdash \left(\!\left| p \mapsto \begin{array}{c} d_i : t_i \\ d_j : f_j \end{array} \right|\!\right) : S \to T}{\Gamma\,,\, p : S \vdash t_i : F_i(A,S) \quad \Gamma\,,\, p : S \vdash f_j : E_j(A) \to F_j(A,S)}$$	$R(A) = T$
map $F(+)$	$$\dfrac{\Gamma \vdash F\{f\} : S \to T}{\Gamma \vdash f : A \to B}$$	$S = F(A)\,,\ F(B) = T$
map $F(-)$	$$\dfrac{\Gamma \vdash F\{f\} : S \to T}{\Gamma \vdash f : A \to B}$$	$S = F(B)\,,\ F(A) = T$
map $F(*)$	$$\dfrac{\Gamma \vdash F\{f \,\&\, g\} : S \to T}{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : B \to A}$$	$S = F(A)\,,\ F(B) = T$
map $F(?)$	$$\dfrac{\Gamma \vdash F\{\_\} : S \to T}{.}$$	$S = F(A)\,,\ F(B) = T$
function variable	$$\dfrac{\Gamma\,,\, f : X \to Y \vdash f : S \to T}{.} \quad S = X\,,\ Y = T \qquad \dfrac{\Gamma\,,\, \rho \vdash f : S \to T}{\Gamma \vdash f : S \to T}$$	$(f \notin \rho)$
function symbol	$$\dfrac{\vdash f : S \to T}{.}$$	$S = X_0(A)\,,\ Y_0(A) = T$
function symbol	$$\dfrac{\Gamma \vdash f\{g_1, \ldots, g_n\} : S \to T}{\Gamma \vdash g_1 : X_1(A) \to Y_1(A) \ \cdots \ \Gamma \vdash g_n : X_n(A) \to Y_n(A)}$$	$S = X_0(A)\,,\ Y_0(A) = T$

1. In the *case* rule the patterns $p_1, \ldots, p_n$ need not be exhaustive. For typechecking to succeed, they need only be of the same type.

2. The *fold* rule illustrates decomposition of fold functions for inductive datatypes. The datatype L is assumed to be of the form

$$\begin{array}{rcl} \text{data } L(A) \to C & = & c_1 : E_1(A, C) \to C \\ & | & \cdots \\ & | & c_n : E_n(A, C) \to C \end{array}$$

   Note that each constructor of the datatype must have a corresponding phrase in the fold function.

3. The *unfold* rule illustrates decomposition of unfold functions for co-inductive datatypes. The datatype R is assumed to be of the form

$$\begin{array}{rcl} \text{data } C \to R(A) & | & d_i : C \to F_i(A, C) \\ & | & d_j : C \to E_j(A) \Rightarrow F_j(A, C) \end{array}$$

   Each destructor must have a corresponding phrase in the unfold. The phrase for each first-order destructor d_i must be a term and the phrase for each higher-order destructor d_j must be a function.

4. The four *map* rules illustrate decomposition of map functions for type constructors with parameters of positive variance $(+)$, negative variance $(-)$, bivariance $(*)$, and non-variance $(?)$. Although these rules are given for a type constructor F of one parameter, we assume the presence of a general rule for type constructors with many parameters of possibly mixed variances.

5. The *function variable* rules match a function symbol f with its latest occurrence in the context. If a function symbol does not occur in the context then it must have been previously declared either by a function definition or in a datatype declaration.

6. The *function symbol* rules illustrate decomposition of a previously declared function with 0 or more function parameters. A function symbol with 0 parameters can be any of:

   (a) a constructor $c_i : E_i(A, L(A)) \to L(A)$ introduced by a datatype declaration of the form

$$\begin{array}{rcl} \text{data } L(A) \to C & = & \cdots \\ & | & c_i : E_i(A, C) \to C \\ & | & \cdots \end{array}$$

(b) a first-order destructor $d_i : R(A) \to F_i(A, R(A))$ or higher-order destructor $d_j : E_j(A) \times R(A) \to F_j(A, R(A))$ introduced by a datatype declaration of the form

$$\begin{array}{lcl} \text{data } C \to R(A) & = & \cdots \\ & | & d_i : C \to F_i(A, C) \\ & | & d_j : C \to E_j(A) \Rightarrow F_j(A, C) \\ & | & \cdots \end{array}$$

(c) a user-defined function with 0 macro parameters:

$$\begin{array}{ll} \text{def} & f : X_0(A) \to Y_0(A) \\ = & \cdots \end{array}$$

A function symbol with more than 0 parameters is assumed to have been introduced by a definition of the form

$$\begin{array}{ll} \text{def} & f\{m_1 : X_1(A) \to Y_1(A), \ldots, m_n : X_n(A) \to Y_n(A)\} : X_0(A) \to Y_0(A) \\ = & \cdots \end{array}$$

4 Solving Equations

The processing of the typechecker can be divided into two parts. The first part (described in the previous section) involves collecting equations of types by decomposing expressions. The second part (described in this section) attempts to solve those equations.

The typechecker employs a unification algorithm which attempts to solve equations of types. Given an equation $S = T$, the algorithm tries to find the most general list of type variable assignments σ such that $S\sigma$ and $T\sigma$ are identical. If such a σ cannot be found then $S = T$ has no solution and the algorithm fails. The algorithm is described by the following functions:

$$
\begin{array}{rcl}
\mathit{check}\,(A/A) & = & [\,] \\
\mathit{check}\,(S/A) & = & [S/A]\ (\mathit{if}\ A\ \mathit{does\ not\ occur\ in}\ S) \\
 & = & \mathit{fail}\ (\mathit{otherwise}) \\
\\
\mathit{match}(A, S) & = & \mathit{check}\,(S/A) \\
\mathit{match}(S, A) & = & \mathit{check}\,(S/A) \\
\mathit{match}(F(\vec{S}), F(\vec{T})) & = & \mathit{flatten}(\mathit{map}\{\mathit{match}\}(\mathit{zip}(\vec{S}, \vec{T}))) \\
\mathit{match}(F(\vec{S}), G(\vec{T})) & = & \mathit{fail}\ (\mathit{if}\ F \neq G) \\
\\
\mathit{coalesce}(S/A, [\,]) & = & [\,] \\
\mathit{coalesce}(S/A, T/A :: \mathit{Rest}) & = & \mathit{append}(\mathit{match}(S, T)\ ,\ \mathit{coalesce}(S/A, \mathit{Rest})) \\
\mathit{coalesce}(S/A, T/B :: \mathit{Rest}) & = & \mathit{append}(\mathit{check}(T[S/A]/B)\ ,\ \mathit{coalesce}(S/A, \mathit{Rest})) \\
\\
\mathit{linearize}([\,]) & = & [\,] \\
\mathit{linearize}(S/A :: \mathit{Rest}) & = & S/A :: \mathit{linearize}(\mathit{coalesce}(S/A, \mathit{Rest})) \\
\\
\mathit{unify}(S, T) & = & \mathit{linearize}(\mathit{match}(S, T))
\end{array}
$$

The function *unify* attempts to solve an equation $S = T$. The algorithm is given in two steps:

1. The *match* function builds a list of type variable assignments

$$[S_1/A_1, \ldots, S_n/A_n]$$

   such that for each $1 \leq i \leq n$, A_i does not occur in S_i.

2. The *linearize* function takes the output of the *match* function and produces a list of assignments

$$[S_1/A_1, \ldots, S_n/A_n]$$

such that $A_1, \ldots, A_n$ are distinct, and for each $1 \le i \le j \le n$, A_i does not occur in any S_j.

If an equation $S = T$ has some solutions, the *unify* function will find the most general of these. Otherwise, *unify* will fail.

As shown in the rules for collecting equations, the typechecker usually needs to solve many equations, rather than just a single equation. This is accomplished incrementally during the equation collection process. The typechecker keeps track of its solution in a list of assignments which gets updated each time a new equation is encountered. Given a solution σ and a new equation $S = T$, the following *add* function shows how the effect of $S = T$ is added to σ:

$$\boxed{\mathit{add}(\sigma, S = T) \quad = \quad \mathit{append}(\sigma, \mathit{unify}(S\sigma, T\sigma))}$$

First the existing substitution σ is applied to $S = T$ so the new equation becomes $S\sigma = T\sigma$. This ensures no variables in σ occur in the solution to the new equation. On success, the result of the *add* function becomes the new solution so far.

When defining a function, the user may specify some typing information about the function being defined. In order to correctly deal with user-specified types the typechecker regards user-specified type variables specially as constants, which we call *user variables*. User variables have the property that they can only be matched with ordinary type variables. The *check* function is extended as follows:

$$\boxed{\begin{array}{rcl} \mathit{check}\,(U/U) & = & [\,] \\ \mathit{check}\,(U/A) & = & [U/A] \\ \mathit{check}\,(\_/U) & = & \mathit{fail}\ (\mathit{some\ user\text{-}specified\ type\ is\ too\ general}) \end{array}}$$

5 ***Hash*** and ***At***

5.1 Typechecking the ***Hash***

The *hash* syntax (#) is an extension of the Charity term logic which allows the phrases of a fold function to directly access the data structure being folded over. For example, the following function inserts an element into a sorted list:

$$\text{def } \mathit{insert}\{\mathit{lt} : A \times A \to \mathit{bool}\} : A \times \mathit{list}(A) \to \mathit{list}(A)$$

$$= (a, L) \mapsto \left\{\!\!\left|\; \begin{array}{rl} \mathit{nil}: & () \mapsto \mathit{cons}(a, \mathit{nil}()) \\ \mathit{cons}: & (b, N) \mapsto \left\{ \begin{array}{l} \mathit{true}() \mapsto \mathit{cons}(a, \mathit{cons}\ \#) \\ \mathit{false}() \mapsto \mathit{cons}(b, N) \end{array} \right\} \mathit{lt}(a, b) \end{array} \;\right|\!\!\right\} L$$

The *hash* symbol (#) can be regarded as a variable which can only be used in the phrases of a fold. The type of each # depends on the phrase in which it occurs. Given an inductive datatype

$$\begin{array}{rcl} \text{data } L(A) \to C & = & c_1 : E_1(A,C) \to C \\ & | & \cdots \\ & | & c_n : E_n(A,C) \to C \end{array}$$

a # in the c_i phrase of a fold over $L(A)$ is of type $E_i(A, L(A))$. The hash is easily implemented in the typechecker by the following new rule for decomposing folds:

$$\boxed{\dfrac{\Gamma \vdash \left\{\begin{array}{|c|} c_1 : f_1 \\ \cdots \\ c_n : f_n \end{array}\right\} : S \to T}{\begin{array}{c} \Gamma\,,\#: E_1(A, L(A)) \vdash f_1 : E_1(A,T) \to T \\ \cdots \\ \Gamma\,,\#: E_n(A, L(A)) \vdash f_n : E_n(A,T) \to T \end{array}}\; S = L(A)}$$

In this way, whenever a # is encountered in a fold phrase, its type will be found by finding the last occurrence of the # variable in the context. Note that the typechecker assumes # only occurs in the phrases of a fold function.

5.2 Typechecking the *At*

The *at* syntax (@) is an extension of the Charity term logic which allows the phrases of an unfold function to exit prematurely from a right datatype recursion.

5.2.1 Example

Given the following datatype of infinite lists:

$$\begin{array}{rcl} \text{data } C \to \mathit{inflist}(A) & = & \mathit{head} : C \to A \\ & | & \mathit{tail} : C \to C \end{array}$$

Consider the problem of inserting an element into its proper position in a sorted infinite list. The following unfold function accomplishes this task:

def $\mathit{pushdown}\ \{\mathit{lt} : A \times A \to \mathit{bool}\} : A \times \mathit{inflist}(A) \to \mathit{inflist}(A)$

$$= (a, I) \mapsto \left(\left|\begin{array}{lrl} b_0(as) \mapsto & \mathit{head}: & \left\{\begin{array}{l} \mathit{true} \mapsto a \\ \mathit{false} \mapsto \mathit{head}(as) \end{array}\right\} \mathit{lt}(a, \mathit{head}(as)) \\ & \mathit{tail}: & \left\{\begin{array}{l} \mathit{true} \mapsto b_1(as) \\ \mathit{false} \mapsto b_0(\mathit{tail}(as)) \end{array}\right\} \mathit{lt}(a, \mathit{head}(as)) \\ b_1(as) \mapsto & \mathit{head}: & \mathit{head}(as) \\ & \mathit{tail}: & b_1(\mathit{tail}(as)) \end{array}\right|\right) b_0(I)$$

In this function, the state $b_0(as)$ indicates the element a has yet to be inserted and the state $b_1(as)$ indicates the element a has already been inserted. Notice that once the state $b_1(as)$ is entered, the unfold simply reconstructs the remaining infinite list as.

Using the @ syntax, the above function can be expressed more concisely as follows:

$$\text{def } \mathit{pushdown}'\{\mathit{lt} : A \times A \to \mathit{bool}\} : A \times \mathit{inflist}(A) \to \mathit{inflist}(A)$$

$$= (a, I) \mapsto \left(\!\!\left| \; as \mapsto \begin{array}{rl} \mathit{head} : & \left\{ \begin{array}{l} \mathit{true} \mapsto a \\ \mathit{false} \mapsto \mathit{head}(as) \end{array} \right\} \mathit{lt}(a, \mathit{head}(as)) \\ \mathit{tail} : & \left\{ \begin{array}{l} \mathit{true} \mapsto @(as) \\ \mathit{false} \mapsto \mathit{tail}(as) \end{array} \right\} \mathit{lt}(a, \mathit{head}(as)) \end{array} \right|\!\!\right) I$$

In this function, the code $@(as)$ indicates that the unfold is to terminate and simply return the remaining infinite list as. This behavior lets us avoid the explicit use of coproducts to give extra state information as in the first function above.

□

5.2.2 Example

Given a positive integer n, consider how to generate an infinite list:

$$n, n-1, \ldots, 1, 0, 0, 0, 0, 0, \ldots$$

whose elements successively decrease by 1 and eventually settle into steady 0's. This list is generated by the following unfold:

$$\left(\!\!\left| \; i \mapsto \begin{array}{rl} \mathit{head} : & i \\ \mathit{tail} : & \left\{ \begin{array}{l} \mathit{true} \mapsto 0 \\ \mathit{false} \mapsto i - 1 \end{array} \right\} i = 0 \end{array} \right|\!\!\right) n$$

Once the state variable i reaches the value 0, the $i = 0$ test in the *tail* phrase will cause the *true* branch of the case function to be taken. This is rather redundant, however, since the $i = 0$ test will always be evaluated, even well after i reaches 0.

Another way to generate the infinite list is to use a coproduct to add information to the state:

$$\left(\!\!\left| \begin{array}{ll} b_0(i) \mapsto & \begin{array}{rl} \mathit{head} : & i \\ \mathit{tail} : & \left\{ \begin{array}{l} \mathit{true} \mapsto b_1() \\ \mathit{false} \mapsto b_0(i-1) \end{array} \right\} i = 0 \end{array} \\ b_1() \mapsto & \begin{array}{rl} \mathit{head} : & 0 \\ \mathit{tail} : & b_1() \end{array} \end{array} \right|\!\!\right) b_0(n)$$

In this unfold once i reaches 0 the state wrapper switches from b_0 to b_1 in order to avoid subsequent redundant evaluations of the $i = 0$ test.

A third way to generate the infinite list is to use the @ syntax as follows:

$$\left(\!\!\left| i \mapsto \begin{array}{rl} \mathit{head}: & i \\ \mathit{tail}: & \left\{ \begin{array}{l} \mathit{true} \mapsto @\left(\!\left| \_ \mapsto \begin{array}{l} \mathit{head}: 0 \\ \mathit{tail}: () \end{array} \right|\!\right) () \\ \mathit{false} \mapsto i - 1 \end{array} \right\} i = 0 \end{array} \right|\!\!\right) n$$

Once the $i = 0$ test evaluates to *true* the outer unfold terminates and the much simpler inner unfold (which generates only 0's) is returned.

□

In order for the typechecker to handle the @, we can modify the unfold decomposition rule in a manner somewhat analogous to the way in which the fold decomposition rule is modified to handle the #. Assume we have the following co-inductive datatype:

$$\begin{array}{rcl} \text{data } C \to R(A) & = & d_i : C \to F_i(A, C) \\ & \mid & d_j : C \to E_j(A) \Rightarrow F_j(A, C) \end{array}$$

A naive modification to the unfold decomposition rule would be to simply add @ as a function symbol to the context as follows:

$$\frac{\Gamma \vdash \left(\!\left| p \mapsto \begin{array}{c} d_i : t_i \\ d_j : f_j \end{array} \right|\!\right) : S \to T}{\begin{array}{rl} \Gamma, p : S, @ : R(A) \to S & \vdash t_i : F_i(A, S) \\ & \vdash f_j : E_j(A) \to F_j(A, S) \end{array}} \; R(A) = T$$

This rule is problematic, however. The idea of adding @ to the context is good since whenever @ is encountered in the phrase of an unfold, its type can be found in the context. Also, provided that @ is used correctly within an unfold (i.e. only in positions of recursion), the above rule is sufficient. The problem arises when @ is used incorrectly (i.e. in a non-recursive position) as this kind of error may not get caught by the typechecker. For example, consider the following nonsense term:

$$\left(\!\left| x \mapsto \begin{array}{r} \mathit{head}: @\mathit{ints} \\ \mathit{tail}: x + 1 \end{array} \right|\!\right) 0$$

Using the above rule, @ would have type $\mathit{inflist}(\mathit{int}) \to \mathit{int}$ and typechecking would not fail as it should for this function.

In order for the typechecker to ensure that @ is only used in recursive positions in the phrases of an unfold we introduce the concept of a @ *exception type*. This special type is introduced by the @ function symbol and can only occur in recursive positions in the types of the phrases of an unfold. The above decomposition rule is modified as follows:

$$\frac{\Gamma \vdash \left(\!\left| p \mapsto \begin{array}{c} d_i : t_i \\ d_j : f_j \end{array} \right|\!\right) : S \to T}{\Gamma\,,\, p : S\,,\, @ : R(A) \to S@R(A) \;\; \begin{array}{l} \vdash t_i : F_i(A, S@R(A)) \\ \vdash f_j : E_j(A) \to F_j(A, S@R(A)) \end{array}} \; R(A) = T$$

The @ *type constructor* is used to explicitly flag the recursive positions in the codomain types of the phrases of an unfold. It is only at these positions that the unfold recursion can either continue through the evolution of state, or terminate through an @ exception. Such two-way behavior is reflected in how the effect of the @ can be implemented by using, for example, a coproduct wrapper to signal a change of state.

Types of the form $C@D$ are called @ *exception types* and are treated specially by the type equation solver. In order to describe this special treatment, we require a notion of exception propagation through types.

Propagating Exceptions Through Types

The general form of a decomposition rule for a function is

$$\frac{\Gamma \vdash c\{f\} : S \to T}{\Gamma \vdash f : P_1(A) \to Q_1(A)} \; S = P_0(A)\,,\, Q_0(A) = T$$

where, based on the structure of c, the sub-expressions f are extracted and recursively decomposed. Furthermore, also based on the structure of c, equations involving the types S and T are introduced as required. Here S and T are regarded as the expected types for the function $c\{f\}$. The equations $S = P_0(A)$ and $Q_0(A) = T$ are introduced to further constrain these types. When the typechecker attempts to solve equations of types, such constraints "propagate" to the result types of an expression to ultimately give a valid typing for an expression or fail due to incompatible constraints.

The essential property of exception types is that they should only propagate towards the codomains of the phrases in an unfold, and not beyond this point (i.e. exception types generated within an unfold should not be visible outside of that unfold). Also, non-exception types should be able to propagate forward through exception types (but not backward).

By regarding equations of types as ordered pairs, we can control the direction of propagation of exceptions. In the decomposition rules for patterns (3.1), terms (3.2), and functions (3.3), we impose an ordering on the type equations by simply replacing $=$ by $\leq$ (in fact, the type equations given there have been oriented correctly). For example, the form of the above general decomposition rule for a function is now

$$\frac{\Gamma \vdash c\{f\} : S \to T}{\Gamma \vdash f : P_1(A) \to Q_1(A)} \; S \leq P_0(A)\,,\, Q_0(A) \leq T$$

The equation solver of the typechecker is modified as well. The *add* function is essentially as before (except now the order matters):

$$\boxed{\mathit{add}(\sigma, S \leq T) \quad = \quad \mathit{append}(\sigma, \mathit{unify}(S\sigma, T\sigma))}$$

and the *match* function is extended to handle exceptions as follows:

$$\boxed{\begin{array}{rcl} \mathit{match}(A, C@D) & = & \mathit{check}\,(C@D/A) \\ \mathit{match}(C@D, A) & = & \mathit{check}\,(C@D/A) \\ \mathit{match}(C@D, C'@D') & = & \mathit{append}(\mathit{match}(C, C'), \mathit{match}(D, D')) \\ \mathit{match}(F(\vec{S}), C@D) & = & \mathit{match}(F(\vec{S}), C) \\ \mathit{match}(C@D, F(\vec{S})) & = & \mathit{fail} \end{array}}$$

The effect of the ordering is to allow exception types to propagate freely through type variables and other exception types, and non-exception types can only propagate forward through exception types.

Finally, we need to ensure that no exception types propagate outside of the unfold in which they are introduced. This check is done after the phrases of an unfold have been successfully typechecked and involves applying the current substitution list to the codomain type and checking that no @ types occur in the result.

5.2.3 Example

This example illustrates how an unfold function with @ in a non-recursive position is caught by the typechecker. Given that *ints* is a term of type *inflist*(*int*), consider typechecking the function:

$$\vdash \left(\!\middle| x \mapsto \begin{array}{r} \mathit{head} : @\mathit{ints} \\ \mathit{tail} : x + 1 \end{array} \middle|\!\right) : A_0 \to A_1$$

If Γ is the context

$$x : A_0\,,\; @ : \mathit{inflist}(A_2) \to A_0@\mathit{inflist}(A_2)$$

then the *head* phrase is decomposed as follows:

$$\dfrac{\dfrac{\Gamma \vdash \mathit{ints} : A_3}{\cdot} \qquad \mathit{inflist}(\mathit{int}) \leq A_3 \qquad \dfrac{\Gamma \vdash @ : A_3 \to A_2}{\cdot} \qquad \begin{array}{l} A_3 \leq \mathit{inflist}(A_2)\,, \\ A_0@\mathit{inflist}(A_2) \leq A_2 \end{array}}{\Gamma \vdash @\mathit{ints} : A_2}$$

The first two equations give substitution list, $[\mathit{inflist}(\mathit{int})/A_3\,,\; \mathit{int}/A_2]$ and the third equation causes a failure because the @ exception cannot propagate through the *int* type.

$\square$

5.2.4 Example

This is an example of an unfold which successfully typechecks using the @. If *zeros* has type *inflist*(*int*) then consider the function:

$$\vdash \left(\!\!\left| i \mapsto \begin{array}{rl} head: & i \\ tail: & \left\{ \begin{array}{l} true \mapsto @zeros \\ false \mapsto i-1 \end{array} \right\} i=0 \end{array} \right|\!\!\right) : A_0 \to A_1$$

If Γ is the context

$$i : A_0 \,,\, @ : inflist(A_2) \to A_0 @ inflist(A_2)$$

then the *tail* phrase decomposes as follows:

$$\dfrac{\Gamma \vdash \left\{ \begin{array}{l} true \mapsto @zeros \\ false \mapsto i-1 \end{array} \right\} i=0 : A_0 @ inflist(A_2)}{\dfrac{\Gamma \vdash i=0 : A_3}{\cdots} \quad bool \leq A_3 \quad \dfrac{\Gamma \vdash \left\{ \begin{array}{l} true \mapsto @zeros \\ false \mapsto i-1 \end{array} \right\} : A_3 \to A_0 @ inflist(A_2)}{\cdots}}$$

The *true* branch of the case function decomposes as follows:

$$\dfrac{\Gamma \,,\, true : A_3 \vdash @zeros : A_0 @ inflist(A_2)}{\dfrac{\Gamma \vdash zeros : A_4}{.} \quad inflist(int) \leq A_4 \quad \dfrac{\Gamma \vdash @ : A_4 \to A_0 @ inflist(A_2)}{.}} \quad \begin{array}{l} bool \leq A_3 \\ \\ A_4 \leq inflist(A_2) \\ A_0 @ inflist(A_2) \leq A_0 @ inflist(A_2) \end{array}$$

and the *false* branch decomposes as follows:

$$\dfrac{\Gamma \,,\, false : A_3 \vdash i-1 : A_0 @ inflist(A_2)}{\dfrac{\Gamma \vdash i : A_5}{.} \quad A_0 \leq A_5 \quad \dfrac{\Gamma \vdash \_ - 1 : A_5 \to A_0 @ inflist(A_2)}{.}} \quad \begin{array}{l} bool \leq A_3 \\ \\ A_5 \leq int \\ int \leq A_0 @ inflist(A_2) \end{array}$$

Notice how in the *true* branch, an exception type is propagated to the codomain of the case, whereas in the *false* branch, an *int* type is propagated to the codomain of the case. Solving the resulting equations for A_0 and A_1 gives $A_0 = int$ and $A_1 = inflist(int)$.

□

6 References

[1] J. P. Jouannaud and C. Kirchner, Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification, Unité Associée au CNRS UA 410: Al Khowarizmi, Rapport de Recherche #561 (1990).

[2] R. Milner, A Theory of Type Polymorphism in Programming, *Journal of Computer and System Sciences* **17** (1978) 348–375.