IMPLICIT GUARD EXPRESSIONS IN
FUNCTIONAL PROGRAMMING LANGUAGES

Nils M Holm, 2014

1. Introduction

Most functional programming languages use *pattern matching* to bind arguments to values. Guard expressions are used in some languages to put semantic constraints on patterns, but patterns are syntactically separated from guard expressions, which reduces locality (i.e. pattern variables and constraints are not in the same place) and unnecessarily duplicates code.

In this paper, the principle of *implicit guard expressions* is introduced, which combines patterns and guard expression in the same syntactic construct, thereby creating locality and eliminating redundant code.

2. Pattern Matching

In the following, a small set of objects will be used to explain pattern matching. This set consists of *numbers*, *k-tuples*, *lists*, and *variables*.

- A *number* is a sequence of decimal digits, i.e. an integer.
- A *k-tuple* (or *tuple*) is an ordered set of *k* elements. E.g., (a,b,c) is a 3-tuple. Tuples may contain any type of object, even tuples. Numbers, lists, and variables are 1-tuples. Parentheses can be omitted around 1-tuples.
- A *list* is a sequence of elements of any type, e.g. [a,b,c]. Elements can be added to lists using the *cons* (::) operator: a :: [b] → [a, b].[1]
- A *variable* is denoted by a sequence of alphabetic characters. Variables are bound to values by pattern matching.

The pattern matching function m maps a pattern p and an object x to a tuple $((v_1, a_1), \ldots)$, indicating that each variable v_i of p is now bound to the corresponding argument a_i of x. For instance:

$$m([a, b], [1, 2]) \ \rightarrow \ ((a, 1), (b, 2))$$

When a match fails, the function returns ε instead:

$$m(1, 2) \ \rightarrow \ \varepsilon$$

A result of () (0-tuple, unit) indicates that the match succeded, but no variables were bound:

$$m(1, 1) \ \rightarrow \ ()$$

The matching function m works as follows:

- Each number matches itself.
- A variable matches any object, binding that object to the variable.
- A k-tuple $T_{k,1}$ matches another k-tuple $T_{k,2}$, if both tuples have pairwise equal elements. Only k-tuples of the same order (number of elements) match. For example:

$$m((1, 2), (1, 2)) \ \rightarrow \ ()$$
$$m((1, 2), (2, 1)) \ \rightarrow \ \varepsilon$$
$$m((1, 2), (1, 2, 3)) \ \rightarrow \ \varepsilon$$

- A list L_1 matches another list L_2, if (1) both lists are the empty list [], or (2) $L_1 = h_1 :: t_1$ and $L_2 = h_2 :: t_2$ and $m(h_1, h_2) \neq \varepsilon$ and $m(t_1, t_2) \neq \varepsilon$. (I.e.: both lists contain pairwise equal elements.)

When matching a list L against a pattern p containing the constructor $::$, the constructor decomposes the value L into the first element of the list and the rest of the list and optionally binds the components of the list to variables. For example:

$$m(1 :: [2], 1 :: [2]) \ \rightarrow \ ()$$
$$m(1 :: [2], [1, 2]) \ \rightarrow \ ()$$
$$m(1 :: x, [1, 2, 3]) \ \rightarrow \ ((x, [2, 3]))$$
$$m(h :: t, [1, 2, 3]) \ \rightarrow \ ((h, 1), (t, [2, 3]))$$

3. Building Functions with Pattern Matching

A tiny subset of the ML programming language[2] will be used to demonstrate how to build funtions with

[1] "a → b" means "a evaluates to b".

pattern matching. Generally, a *function* f of patterns $p_1, \cdots$ to expressions $x_1, \cdots$ is defined as follows:

$$fun\ f\ p_1 = x_1 \mid \cdots$$

where the vertical bar denotes a logical "or". In the function application

$$f\ a$$

the argument a is matched against each pattern p_i of f, and the expression x_i associated with the first matching pattern is evaluated with the bindings established by m in effect. For example, the following program implements the power function (x raised to the power of y):

$$\begin{aligned} &fun\ P\ (x, 0) = 1 \\ &\quad \mid (x, y) = x * P(x, y-1) \end{aligned}$$

4. Guarded Patterns

A *guard* is an expression that belongs to the pattern part rather than the expression part of a function. It puts some semantic constraints on the values a pattern will match. The problem guards solve is as follows:[3]

$$\begin{aligned} &fun\ gcd\ (a, 0) = a \\ &\quad \mid (0, b) = b \\ &\quad \mid (a, b) = \text{if } a < b\ then \\ &\qquad\qquad gcd\ (a, b\ mod\ a) \\ &\qquad\quad else \\ &\qquad\qquad gcd\ (b, a\ mod\ b) \end{aligned}$$

Here the otherwise declarative nature of the program is disturbed by making the distinction between the cases $a < b$ and $a \geq b$ on the *expression* side of the function. In a mathematical definition, though, the distinction would be made like this:

$$gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ gcd(a, b\ mod\ a) & \text{if } a < b \\ gcd(b, a\ mod\ b) & \text{otherwise} \end{cases}$$

I.e. there is only a pattern and an expression side, and nothing "in between", like the "if" in the above program. Using guard expressions, the mathematical defintion can be translated directly to a corresponding program:[4]

$$\begin{aligned} &fun\ gcd\ (a, 0) = a \\ &\quad \mid (0, b) = b \\ &\quad \mid (a, b)\ where\ a < b \\ &\qquad\quad = gcd\ (a, b\ mod\ a) \\ &\quad \mid (a, b) = gcd\ (b, a\ mod\ b) \end{aligned}$$

Guards also allow to use the same basic (unguarded) pattern multiple times, but each time with different constraints.

In the above extension of the ML language,[5] the conditional expression c after the *where* keyword limits the case of the function to values that match the pattern *and* satisfy the condition c. So the third case of *gcd* is only selected when $a < b$ holds. Otherwise, the final case, without any limiting guard, is chosen.

The guard mechanism described here so far will be called an *explicit guard*, because it uses an explicit expression, following the *where* keyword, to formulate the limiting condition.

While explicit guards are a general and flexible mechanism for specifying guard expressions, they can be cumbersome in some simple cases. For instance:

$$\begin{aligned} &signum\ x\ where\ x < 0 = \tilde{}1 \\ &\quad \mid x\ where\ x > 0 = 1 \\ &\quad \mid x = 0 \end{aligned}$$

5. Implicit Guards

Under certain conditions, the guard expressions can be pulled into the pattern, so that variables of the guard expression become variables of the pattern at the same time. This will be called an *implicit guard*, because the guard is contained implicitly in the pattern.

Using implicit guards, the *signum* function can be written this way:

---

[2] See "The Definition of Standard ML" by Milner, et al, MIT Press 1997.

[3] The *gcd* function computes the greatest common divisor of a and b using the Euclidean algorithm.

[4] The *where* syntax is not part of Standard ML.

[5] See "The mLite Language" by Holm, 2014

$$\begin{array}{l} signum\ x < 0 = \tilde{}1 \\ \quad\quad\ \ \mid x > 0 = 1 \\ \quad\quad\ \ \mid x \quad\ \ = 0 \end{array}$$

I.e. "signum *of* $x < 0$ is -1", etc.

Implicit guards allow to apply conditions *on the spot* inside of a pattern, which is more concise than explicit guards and increases the locality of the guard expressions, because they do not have to be moved to a separate clause.

Here is another example:[6]

$$\begin{array}{l} fun\ \ filter\ (f, []) = [] \\ \quad\quad\quad\quad \mid (f, f\ \ h :: t) = h :: filter(f, t) \\ \quad\quad\quad\quad \mid (f, h :: t) = filter(f, t) \end{array}$$

The *filter* function extracts objects satisfying the property f from a list, e.g.:[7]

$$filter((fn\ \ x = x < 0), [1, \tilde{}2, 3, \tilde{}4, 5]) \ \rightarrow\ [\tilde{}2, \tilde{}4]$$

It uses the implicit guard $f\ h$ in the pattern $(f, f\ h :: t)$ to select matching elements from the list. Because function application binds stronger than the :: operator, the predicate f is applied to the first element of the list, and the pattern matches only if $f\ h$ returns truth.

While implicit guards are more concise and more local than explicit guards, they do not offer the same degree of flexibility.

These are the conditions under which implicit guards may be used:

- No two variables of the same pattern may be combined in the same guard.
- Only one variable may appear in an infix expression in a guard.
- In a function application, the pattern variable must be the rightmost component.
- Multiple implicit guard expressions may appear in a tuple or list, as long as they are independent and obey the above rules.

For example, due to the first rule, the explicit guard

$$(a, b)\ where\ a < b$$

in the *gcd* function cannot be converted to an implicit guard, because it combines the pattern variables a and b in the expression $a < b$.

Implicit guards of the form $a\ R\ b$, where R is an infix operator and both a and b are variables, cannot be used, because it is not possible to determine which one of the variables would be the pattern variable.[8]

In a function application, the rightmost component is considered to be the pattern part. In all of the expressions $f\ x$, $f\ g\ x$, and $f\ (g\ x)$, the x would be the pattern variable.

Note: Because function application has the same syntax as currying and the equal operator (=) is used to separate function bodies from patterns in ML, implicit guards using the = operator or function application have to be parenthesized:

$$fun\ f\ x = y \quad \text{equals} \quad fun\ f = fn\ x = y$$

$$fun\ (f\ x) = y \quad \text{equals} \quad fun\ x\ where\ f\ x = y$$

$$fun\ f\ x = 0 = y \quad \text{equals} \quad fun\ x = (0 = y)$$

$$fun\ f(x = 0) = y \quad \text{equals} \quad fun\ x\ where\ (x = 0) = y$$

6. Implementation

Each implicit guard can be rewritten as a pattern and an explicit guard:

$$f\ h :: t \ = \ h :: t\ where\ f\ h$$

This conversion will be formalized in the following by devising a function ρ of an implicit guard to a tuple of a pattern and an explicit guard expression:

$$\rho(implicit) \ \rightarrow\ (pattern, explicit)$$

Furthermore, the functions P and G will be used to refer to the pattern and guard part of the resulting tuple:

$$P(pattern, guard) \ \rightarrow\ pattern$$

$$G(pattern, guard) \ \rightarrow\ guard$$

6 Of course, this implementation is naïve, it only serves the purpose of demonstrating the use of an implicit guard.

7 The syntax $fn\ \ p = x$ defines an anonymous function of a pattern p to an expression x.

8 One of the variables could be chosen by convention, but even then, more complex formulae would render this approach impractical.

So, for instance:

$$\rho(x < 0) \;\rightarrow\; (x, x < 0)$$
$$P(\rho(x < 0)) \;\rightarrow\; x$$
$$G(\rho(x < 0)) \;\rightarrow\; x < 0$$

The ρ function is defined as follows:

$$\rho(number) \;\rightarrow\; (number, \varnothing)$$
$$\rho(variable) \;\rightarrow\; (variable, \varnothing)$$

The empty set sign $\varnothing$ denotes the absence of a guard expression in a tuple. In some contexts, it can be interpreted as

$$G(p, \varnothing) \;\rightarrow\; true$$

I.e., the tuple $(p, \varnothing)$ indicates an unguarded pattern p.

The most simple case of function application is the combination $(\cdot)$ of two variables:

$$\begin{aligned} \rho(f\ x) &= \rho(f) \cdot \rho(x) \\ &= (f, \varnothing) \cdot (x, \varnothing) \\ &= (x, f\ x) \end{aligned}$$

Note that it is always the pattern of the *righthand side* of $(\cdot)$ that determines the pattern component of the result. More generally, the conversion of function application is defined as:

$$\rho(x\ y) \;=\; (P(\rho\ y), x'\ y')$$

where

$$x' = \begin{cases} G(\rho x) & \text{if } G(\rho x) \neq \varnothing \\ P(\rho x) & \text{otherwise} \end{cases} \tag{C1}$$

and

$$y' = \begin{cases} G(\rho y) & \text{if } G(\rho y) \neq \varnothing \\ P(\rho y) & \text{otherwise} \end{cases} \tag{C2}$$

So whenever a guard is available, it will be used and otherwise the corresponding pattern will be used. A pattern is always available, because $\varnothing$ never appears in a the pattern position of a tuple returned by ρ.

Using the above rules and left-associative function application, higher order application becomes:

$$\begin{aligned} \rho(f\ g\ x) &= \rho(f\ g) \cdot \rho x \\ &= (g, f\ g) \cdot (x, \varnothing) \\ &= (x, (f\ g) x) \end{aligned}$$

Function composition becomes:

$$\begin{aligned} \rho(f\ (g\ x)) &= \rho f \cdot \rho(g\ x) \\ &= (f, \varnothing) \cdot (x, (g\ x)) \\ &= (x, f(g\ x)) \end{aligned}$$

So, generally:

$$\rho(f_1 f_2 \cdots x) = (x, ((f_1 f_2) \ldots) x)$$
$$\rho(f_1 (f_2 \cdots x)) = (x, (f_1 (f_2 \cdots x)))$$

The general forms of infix operation are the following (R denotes an infix operator):

$$\begin{aligned} \rho(id\ R\ x) &= (id, (id\ R\ x)) \\ \rho(x\ R\ id) &= (id, (x\ R\ id)) \\ \rho(id\ R\ id) &= (id, (id\ R\ id)) \\ \rho(id_1\ R\ id_2) &= error \end{aligned}$$

That is, an identifier (variable) may appear on either side of the operator, but if it appears on both sides, it must be the same identifier. Two different identifiers in an infix operation are an error.

More complex operands to infix operators, including chains of infix operations, are covered as follows:

$$\rho(x\ R\ y) \;=\; (p, x'\ R\ y')$$

where

$$p = \begin{cases} P(\rho x) & \text{if } P(\rho x) \text{ is an identifier} \\ P(\rho y) & \text{otherwise} \end{cases}$$

and x' and y' are formed as described in (C1) and (C2).

Naturally, precedence and associativity rules of infix operators have to be obeyed. In the following, R_n denotes a left-associative infix operator of precedence n, where larger values denote stronger binding to operands. Then,

$$\rho(x\ R_1\ y\ R_1\ z) = \rho((x\ R_1\ y)\ R_1\ z)$$
$$\rho(x\ R_2\ y\ R_1\ z) = \rho((x\ R_2\ y)\ R_1\ z)$$
$$\rho(x\ R_1\ y\ R_2\ z) = \rho(x\ R_1\ (y\ R_2\ z))$$

R_R denotes a right-associative operator:

$$\rho(x\ R_R\ y\ R_R\ z) = \rho(x\ R_R\ (y\ R_R\ z))$$

Parentheses override associativity and precedence:

$$\rho((x\ R_1\ y)\ R_2\ z) = \rho((x\ R_1\ y)\ R_2\ z)$$
$$\rho(x\ R_2\ (y\ R_1\ z)) = \rho(x\ R_2\ (y\ R_1\ z))$$
$$\rho(x\ R_1\ (y\ R_1\ z)) = \rho(x\ R_1\ (y\ R_1\ z))$$
$$\rho((x\ R_R\ y)\ R_R\ z) = \rho((x\ R_R\ y)\ R_R\ z)$$

Without parentheses function application binds stronger than infix operators:

$$\rho(f\ x+1) = \rho((f\ x)+1)$$

Of course, this means that an actual implementation has to consider the properties of pre-defined operators and, in the case of extensible languages like ML, also the properties of user-defined operators.

In the context of implicit guard conversion, logic operators, such as ML's *orelse* and *andalso*, can be considered to be low-precedence infix operators.

More high-level constructs, like *if-then-else*, are best kept in function bodies, and should not be used in guards anyway, because they would blur the border between guards and function bodies. Guard expressions, both implicit and explicit, should be short and easily comprehensible forms.

k-tuples are converted as follows:

$$\rho(x_1, x_2, \ldots) = (p, g)$$

where

$$p = (P(\rho x_1), P(\rho x_2), \ldots)$$

and

$$g = (G(\rho x_1)) \times (G(\rho x_2)) \times \cdots$$

and

$$G(x, \varnothing) = true^9$$

Here $\times$ denotes the logical "and" operation, i.e. the explicit guard of a tuple is the conjunction of all implicit guards in the tuple.

Lists are converted in the same way as tuples:

$$\rho[x_1, x_2, \ldots] = (p, g)$$

where

$$p = [P(\rho x_1), P(\rho x_2), \ldots]$$
$$g = (G(\rho x_1)) \times (G(\rho x_2)) \times \cdots$$

Constructors like ::, finally, are part of the pattern, but their arguments may contain implicit guards. Hence:

$$\rho(x_1 :: x_2) = (p, g)$$

where

$$p = P(\rho x_1) :: P(\rho(x_2))$$
$$g = (G(\rho x_1)) \times (G(\rho(x_2)))$$

7. Examples

$$\begin{aligned}\rho(f\ x = g\ x) &= \rho(\rho(f\ x) = \rho(g\ x)) \\ &= \rho((x, f\ x) = (x, g\ x)) \\ &= (x, f\ x = g\ x) \\ &= x\ where\ f\ x = g\ x\end{aligned}$$

$$\rho[x > 0, y\ mod\ 2 = 0] = (p, g)$$

where

$$\begin{aligned}p &= [P(\rho(x > 0)), P(\rho(y\ mod\ 2 = 0))] \\ &= [x, y]\end{aligned}$$

and

$$\begin{aligned}g &= (G(\rho(x > 0))) \times (G(\rho(y\ mod\ 2 = 0))) \\ &= (x > 0) \times (y\ mod\ 2 = 0)\end{aligned}$$

$$\begin{aligned}\rho(f\ h :: t) &= (h :: t, (f\ h) \times true) \\ &= (h :: t, f\ h)\end{aligned}$$

9 Of course, an actual implementation would omit segments of the form "$\times$ *true*", because $x \times true = x$.

8. Conclusion

In functions where independent constraints are placed on individual variables of a pattern, implicit guard expressions deliver the desired improvement: they eliminate redundant code and increase locality, because implicit guard expressions are part of the pattern. In more complex cases, explicit guard expressions must be used.

All implicit guard expressions can be rewritten to explicit ones by the algorithm presented in this paper, making it easy to extend languages that already provide explicit guards.

Work to be done includes:

- Special care has to be taken when combining currying with implicit guard patterns, because a compiler has to identify common patterns in functions like

  $$\begin{aligned} &\mathit{fun}\ p\ x\ 0 = 1 \\ &\quad\mid x\ y = p\ x\ (y-1) \end{aligned}$$

  If the pattern x would contain implicit guards in this case, the compiler would still have to be able to identify the common pattern x in order to generate a properly curried function. The mLite language currently solves this problem by not allowing the combination of currying and alternative patterns, but this is to be improved.

- Multiple identifiers in infix expressions should be dealt with in some meaningful way. One option would be to create a tuple of two identifiers, so that the implicit guard $(a < b)$ would translate to (a, b) *where* $(a < b)$. However, this approach looks rather uninituitive. Other alternatives have to be examined.

- The conversion algorithm is to be adapted to syntactical constructs not covered here, like more complex (multi-argument) constructors, etc.

- As the guard conversion algorithm creates guard expressions in conjunctive normal form,[10] the resulting guard expressions may be suitable for automatic ordering by specificity, allowing to formulate programs in a truly declarative form, where the order of clauses in a pattern matching function does not matter. This is to be investigated.

[10] Logic operators may appear in clauses in the implementation described here, so some additional conversion may be necessary.