Subexpression Names

Labels and Labeled Subexpression Names

Any subexpression of a definition can be labeled. The syntax of a labeled expression is

$$label :: expression$$

(The symbol "::" is typed "::".) The label applies to the largest possible expression that follows it. In other words, the end of the labeled expression is the same as the end of the expression that you would get by replacing the "*label* ::" with "$\forall x :$". However, the expression is illegal if removing the label would change the way the expression is parsed. For example,

$$a + lab :: b * c$$

is legal because it is parsed as $a + (lab :: (b * c))$, which is how it would be parsed if the label lab were not there. However,

$$a * lab :: b + c$$

is illegal because it would be parsed as $a * (lab :: (b + c))$ and removing the label causes the expression to be parsed as $(a * b) + c$.

Label parameters are required if labels occur within the scope of bound identifiers. Here is an example.

$$\begin{array}{lcl} F(a) & \triangleq & \forall b : l1(b) :: (a > 0) \Rightarrow \\ & & \qquad\qquad\qquad\quad \land \ldots \\ & & \qquad\qquad\qquad\quad \land\ l2 :: \exists c : \land \ldots \\ & & \qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ \exists d : l3(c, d) :: a - b > c - d \end{array}$$

For this example, $F(A)!l1(B)!l2!l3(C, D)$ names the expression $A-B > C-D$. Note how the parameters of each label are the bound identifiers introduced between it and the next outer-most label. Those identifiers can appear in any order. For example, if the label $l3(c, d)$ were replaced by $l3(d, c)$, then $F(A)!l1(B)!l2!l3(C, D)$ would name the expression $A - B > D - C$.

In this example, a reference to the subexpression labeled by $l3(c, d)$ from outside the definition of F, must specify the values of all the bound identifiers a, b, c, and d. That's why labels must include the bound identifiers as parameters. Also observe that to name a labeled subexpression, we have to name all the labeled subexpressions within which it lies. We're not even allowed to eliminate the label $l2$, even though it is superfluous in this example.

Label names do not conflict with operator names. In this example, any one of the label names $l1$, $l2$, or $l3$ could be replaced by F. The rule for name conflict is the obvious one needed to guarantee that there's no ambiguity in a subexpression name (where we are not allowed to use the number of parameters to disambiguate). Thus, we cannot label the first conjunct of the $\exists\, c$ expression with $l3(c)$, but we could label it with $l1(c)$ or $l2(c)$.

For subexpressions of the definition of an infix, postfix, or prefix operator, we use the "nonfix" form. For example, a subexpression of the definition of && would have the form $\&\&(A, B)\,!\ldots$.

We can also name subexpressions of definitions in instantiated modules. For example, if we have

$$Ins(x) \;\triangleq\; \text{INSTANCE } M \text{ WITH } \ldots$$

and ν is the name of any subexpression of a definition in module M, then $Ins(exp)!\nu$ is the name of the subexpression of the instantiated definition obtained when *exp* is substituted for x.

We call a subexpression name having one of the forms described here a *labeled subexpression name.* We include in this category the trivial case in which there is no label name, only the name of a defined operator—possibly in an instantiated module. The precise definition is contained in the "fine print" below. You probably don't want to read it.

The Fine Print

Here is the general definition explained above with examples. We say that label $lab1$ is the *containing label* of $lab2$ iff (i) $lab2$ lies within the expression labeled by $lab1$ and (ii) if $lab2$ lies within the expression labeled by any other label, then $lab1$ also lies within that expression.

We use the notation that $f(e_1, \ldots, e_k)$ denotes f when $k = 0$. A label lab has the form $id(p_1, \ldots, p_k)$ where id and the p_i are identifiers, the p_i are all distinct, and $\{p_1, \ldots, p_k\}$ is the set of all bound identifiers p_i such that:

- Label lab lies within the scope of p_i.
- If lab has a containing label $labc$, then the expression that introduces p_i lies within the expression labeled by $labc$.

We call id the *name* of the label. Two labels that either have no containing label or have the same containing label must have different names.

A *simple labeled subexpression name* of a module M has the form $prefix\,!\,labexp_1\,!\ldots!\,labexp_n$, where *prefix* has the form $Op(e_1, \ldots, e_{k[0]})$, each $labexp_i$ has the form $id_i(e_1, \ldots, e_{k[i]})$, Op and the id_i are identifiers, and the e_j are expressions. It must satisfy:

- The definition
  $$Op(p_1, \ldots, p_{k[0]}) \;\triangleq\; \ldots$$
  occurs at the top level (not inside a LET or inner module) of M.
- id_1 must be the name of a label lab_1 in the definition of Op that has no containing label.
- If $i > 1$, then id_i must be the identifier of a label lab_i whose containing label is lab_{i-1}.

- $k[i]$ must equal the number of parameters in lab_i, for each $i > 0$.

This labeled subexpression name denotes the expression obtained from the expression labeled with lab_n by substituting for each parameter of Op and of each lab_i the corresponding argument of *prefix* and $labexp_i$, respectively.

A *labeled subexpression name* of a module M is either a simple labeled subexpression name of M or else has the form $Id(e_1, \ldots, e_k)\,!\,\lambda$ where there is a statement

$$Id(e_1, \ldots, e_k) \;\triangleq\; \text{INSTANCE } N \ldots$$

at the outermost level of M and λ is a labeled subexpression name of module N.

Positional Subexpression Names

Instead of using labels, we can name subexpressions of a definition by a sequence of *positional selectors* that indicate the position of the subexpression in the parse tree. Consider this example

$$\begin{array}{lll} F(a) & \triangleq & \land \ldots \\ & & \land \ldots \\ & & \land\ Len(x[a]) > 0 \\ & & \land \ldots \end{array}$$

Here are how some of the subexpressions of this definition are named, where A is an arbitrary expression:

- $F(A)!3$ names $Len(x[A]) > 0$, the third conjunct of $F(A)$—that is, of the right-hand side of the definition with A substituted for a. We think of this conjunct list as the application of a conjunction operator that takes four arguments, the third being $Len(x[A]) > 0$.

- $F(A)!3!1$ names $Len(x[A])$, the first argument of $>$, the top-level operator of the expression $F(A)!3$

- $F(A)!3!1!1$ names $x[A]$, the first (and only) argument of the top-level operator of the expression $F(A)!3!1$.

- The naming of subexpressions of $x[A]$ is based on the realization that this expression represents the application of a function-application operator to the two arguments x and A. Thus, $F(A)!3!1!1!1$ names x and $F(A)!3!1!1!2$ names A

The positional selector "$!\langle$" is always synonymous with $!1$, and "$!\rangle$" is synonymous with $!2$ when selecting the second argument of an operator that takes two arguments. Thus, instead of $F(A)!3!1!1!2$, we could write $F(A)!3!\langle!\langle!\rangle$ or $F(A)!3!\langle!1!\rangle$ or $F(A)!3!1!\langle!2$ or .... As usual, "$\langle$" is typed "<<" and "$\rangle$" is typed ">>".

The use of positional selectors to pick an argument of an operator is self-evident for most operators that do not introduce bound identifiers. Here are the cases that are not obvious.

- In $[f \text{ EXCEPT } ![a] = g, ![b].c = h]$ we select f with $!1$, g with $!2$, and h with $!3$. No other subexpressions of the EXCEPT construct can be named.
- $r.\mathit{fld}$ is an application of a record-field selector operator to the two arguments r and "fld", so $!1$ selects r. (You can also use $!2$ to select "fld", but there's no reason to name a simple string constant with a subexpression name.)
- In $[\mathit{fld}_1 \mapsto \mathit{val}_1, \ldots, \mathit{fld}_n \mapsto \mathit{val}_n]$ and $[\mathit{fld}_1 : \mathit{val}_1, \ldots, \mathit{fld}_n : \mathit{val}_n]$ the selector $!\,i$ names the subexpression val_i for $i \in 1\,..\,n$. The field names fld_i cannot be selected. (There is no point naming fld_i, since it's just a string constant.)
- In $\text{IF } p \text{ THEN } e \text{ ELSE } f$ the selector $!1$ names p, the selector $!2$ names e, and the selector $!3$ names f.
- In $\text{CASE } p_1 \rightarrow e_1 \,\square\, \ldots \,\square\, p_n \rightarrow e_n$ the selector $!i!1$ names p_i and $!i!2$ names e_i. If p_n is the token OTHER, then it cannot be named.
- In $\text{WF}_e(A)$ and $\text{SF}_e(A)$ the selector $!1$ names e and $!2$ names A.
- In $[A]_e$ and $\langle A \rangle_e$ the selector $!1$ names A and $!2$ names e.
- In $\text{LET } \ldots \text{ IN } e$ the selector $!1$ names e. This is rather subtle because we are naming an expression that contains operators defined in the LET clause that are not defined in the context in which the subexpression name appears. Consider this example

  $$\begin{array}{lcl} F & \triangleq & \text{LET } G \triangleq 1 \text{ IN } G + 1 \\ G & \triangleq & 22 \\ H & \triangleq & F!1 \end{array}$$

  The $F!1$ in the definition of H names the expression $G + 1$ in which G has the meaning it acquires in the LET definition. Thus, H is equal to 2, not to 23.

  We will see below how to name subexpressions of LET definitions, such as the first (local) definition of G above.

I now describe selectors for subexpressions of constructs that introduce bound identifiers. Consider this example:

$$R \triangleq \exists x \in S, y \in T : x + y > 2$$

- $R!(X, Y)$ names $X + Y > 2$, for any expressions X and Y.
- $R!1$ names S.
- $R!2$ names T.

In general, for any construct that introduces bound identifiers:

- $!(e_1, \ldots, e_n)$ selects the body (the expression in which the bound identifiers may appear) with each expression e_i substituted for the i^{th} bound identifier.
- If the bound identifiers are given a range by an expression of the form "$\in S$", then $!i$ selects the i^{th} such range S.

For example, in the expression

$$[x, y \in S,\ z \in T \mapsto x + y + z]$$

the selector $!1$ names S, the selector $!2$ names T, and the selector $!(X, Y, Z)$ names $X + Y + Z$.

Parentheses are "invisible" with respect to naming. For example, it doesn't matter if ν names the subexpression $a + b$ or the subexpression $((a + b))$; in either case, $\nu!\langle$ names a.

We usually don't need to name the entire expression to the right of a "$\triangleq$" because the operator being defined names it. However, as observed in Section 16.2.3$^{\square}$, this is not true for recursively defined operators. If *Op* is recursively defined by

$$Op(p_1, \ldots, p_k) \;\triangleq\; exp$$

then "$Op(P_1, \ldots, P_k)!:$" names *exp* with P_i substituted for p_i, for each i in $1\,..\,k$.

A *positional subexpression name* consists of a labeled subexpression name (defined in Section above) followed by a sequence of positional selectors. For example, in

$$F(c) \;\triangleq\; a \,*\, lab :: (b + c * d)$$

$F(7)!lab!\rangle$ names $7*d$. Remember that a labeled subexpression need not contain labels—for example, $F(7)$ is a labeled subexpression name.

Subexpressions of LET Definitions

If a positional subexpression name ν names a LET/IN expression and Op is an operator defined in the LET clause, then $\nu ! Op(e_1, \ldots, e_n)$ is the name of the expression $Op(e_1, \ldots, e_n)$ interpreted in the context determined by ν. For example, in

$$
\begin{array}{lll}
F(a) & \triangleq & \land \ldots \\
 & & \land \text{LET } G(b) \triangleq a + b \\
 & & \quad \text{IN } \ldots
\end{array}
$$

$F(A)!2!G(B)$ names the expression $G(B)$, where the definition of G is interpreted in a context in which A is substituted for a. This expression of course equals $A + B$. (However, if G were recursively defined, $F(A)!2!G(B)$ might not be so simply related to the expression to the right of the "$\triangleq$" in G's definition.) We can also name subexpressions of the definition of G. For example, $F(A)!2!G(B)!\rangle$ names B. The naming process can be continued all the way down, naming subexpressions of LET definitions contained within LET definitions contained within . . . .

If the LET/IN expression is labeled, then it can be named by a labeled subexpression name λ. In that case, $\lambda ! Op(e_1, \ldots, e_n)$ is a labeled subexpression name that names a subexpression of the IN clause with label $Op(p_1, \ldots, p_n)$. To refer to the operator Op defined in the LET clause, just add a "!:" to the end of λ, writing $\lambda !:! Op(e_1, \ldots, e_n)$. In particular, if H is defined to equal the LET/IN expression, then we write $H!:!Op(e_1, \ldots, e_n)$, even if H is not recursively defined.

Subexpressions of an ASSUME/PROVE

If we have

$$\text{THEOREM } Id \triangleq \text{ ASSUME } A_1, \ldots, A_n \text{ PROVE } G$$

then Id is not an expression and cannot be used as one. Subexpressions of an ASSUME/PROVE can be named with labels or positionally, where $Id!i$ names A_i if $1 \leq i \leq n$, and $Id!n{+}1$ names G. However, the assumptions can contain declarations like NEW C, so it is possible to name a subexpression of an ASSUME/PROVE that contains identifiers declared within the ASSUME/PROVE. Such a name can be used only within the scope of those declarations. For example, consider

$$
\begin{array}{l}
\text{THEOREM } T \triangleq \text{ ASSUME } \; x > 0, \text{ NEW } C \in Nat, \; y > C \\
\qquad\qquad\qquad\;\; \text{PROVE } \quad x + y > C \\
\quad \vdots \\
Foo \triangleq \ldots
\end{array}
$$

Then $T!1$ names the expression $x > 0$, which can be used in the definition of *Foo*. However, $T!3$ names the expression $y > C$ that contains the constant C, and the definition *Foo* is not within the scope of the declaration of C, so $T!3$ cannot be used within the definition of *Foo*. In fact, $T!3$ can be used only within the proof of T.

Using Subexpression Names as Operators

Subexpression names can be used as operator names by replacing every part of the form $!id(e_1, \ldots, e_n)$ by $!id$, and every selector $!(e_1, \ldots, e_n)$ by $!@$. For example, consider:

$$\begin{array}{l} F(Op(\_,\_,\_)) \ \triangleq \ Op(1,2,3) \\ G \ \triangleq \ \forall x : P \subseteq \{\langle x,\ y+z \rangle : y \in S,\ z \in T\} \end{array}$$

Then $G!(X)!\rangle!(Y,Z)$ is the expression $\langle X,\ Y+Z \rangle$, so $G!@!\rangle!@$ is the operator

$$\text{LAMBDA } x,\ y,\ z : \langle x,\ y+z \rangle$$

and $F(G!@!\rangle!@)$ equals $\langle 1,\ 2+3 \rangle$.

CLOSE