Specification of a Program Derivation Editor

M. Butler, E. Hedman, P. Nilsson,
R. Rukšėnas, M. Waldén, Y. Zhao
Dept. of Computer Science, Åbo Akademi, Finland

December 1994

Abstract

In this paper, a tool for structuring and manipulating formal program derivations is specified using the Z [13] notation. A program derivation style based on (transitive) relations between programs, as found in the refinement calculus [1, 8, 9], is assumed. The structuring and manipulation of derivations is based on the notion of *refinement diagrams* proposed by Back [3]. This allows for a style of derivation that is much more flexible than top-down refinement and is more suited to large-scale program development. The original refinement diagram notion is also extended with ideas from *window inference* [12] which provides a powerful way of dealing with the use of different relations in sub-derivations.

1 Introduction

Stepwise refinement is a widely used method for formal program derivation. The idea is to start with an abstract specification of a program and transform it by a series of correctness-preserving steps into an executable program. An important feature of stepwise refinement is the ability to refine components of a specification separately, thus reducing the detail involved in individual steps. The refinement calculus, originally proposed in [1] and further developed in [2, 8, 9], is one such approach. In the refinement calculus, specifications are simply high-level programs that are usually non-executable, and correctness is preserved by ensuring that the refinement relation holds between successive programs of the derivation. To allow for separate refinement of components, the operators used to compose programs should preserve the refinement relation.

A problem with the stepwise refinement paradigm is the amount of detail in program derivation. The process involves a large number of steps and is therefore both tedious and error–prone. It is widely accepted that tool support is necessary to overcome these difficulties. Such tools should help ensure the correct application of transformation rules in refinement steps. They should also help the programmer (or programming team) to manage and structure the large collection of program components, and the relationships between them, that arise during the derivation of large programs, keeping track of what has been done and what remains to be done. Existing tools, both prototype and commercial, go some way towards providing such support [16, 11, 6]. But the existing tools tend to assume a top-down approach to program

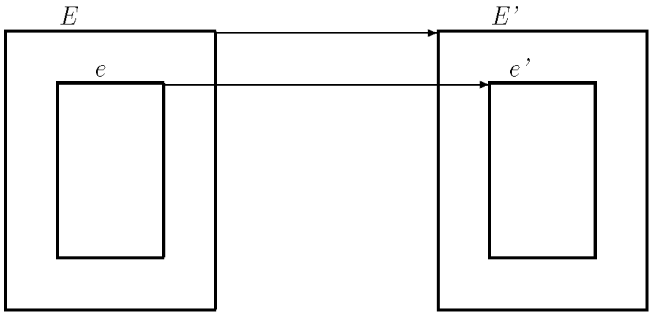

Figure 1: Example Refinement Diagram

derivation. In practice, it is much more productive to be able to move more freely between abstraction levels in either direction, maybe incorporating components developed at lower levels into higher-levels, being able to try out alternative versions of component refinements, and so on.

Back [3] proposed the notion of *refinement diagrams* as a means of modelling the required flexibility in program derivation. A refinement diagram is a graphical way of modelling two forms of relationship between program components: the transformation relation (e.g., program refinement) and the subcomponent relation. For example, Figure 1 is a refinement diagram showing that E is transformed to E', e is transformed to e', e is a subcomponent of E, and e' is a subcomponent of E'. [3] describes several ways in which such diagrams may be manipulated, and those manipulations are formalised in our specification. A prototype tool based on refinement diagrams is described in [4].

A weakness of the ideas proposed in [3] is that only one form of transformation relation is assumed. In practice, the relation that holds between components E and E' may be different to the relation that holds between subcomponents e and e'. For example, consider the program $x := x'.R(x, x')$ which updates program variable x in a way that satisfies the nondeterministic relation R. This program may be refined by refining R to some relation Q. However, in order to preserve the program refinement relation between these programs, we need to ensure that relational inclusion, rather than program refinement, holds between Q and R. In this paper, we label the refinement arrows with appropriate relations (see Figure 2).

In our specification, we allow for different relations in derivations and in order to ensure consistency between these relations we use a a mechanism provided by the *window inference* style of reasoning [12]. In window inference, an expression E is transformed so that a reflexive, transitive relation R is preserved. Attention may be focused on a subexpression e of E, transforming e so that some relation r is preserved. Such focusing is justified by an inference rule of the form:

$$\frac{e \ r \ e'}{E[\ldots e \ldots] \ \ R \ \ E[\ldots e' \ldots]}$$

Window inference also includes the notion of a context in which relations between expressions hold. While it is important for program derivation, we omit the notion of context from this paper for simplicity, but note that our specification can easily be extended to deal with contexts.

The motivation for developing a formal specification of a derivation tool should be obvious. The most immediate benefit has been to clarify our requirements and provide a common understanding of what we require of a program derivation editor. Secondly, it serves as a basis for the correct implementation of the tool.

We assume that programs are represented by expressions, and that the editor manipulates these expressions and the relationships between them in various ways. We start the presentation by specifying expressions and some general manipulations on them. This is followed by the specification of a simple linear derivation editor that allows for top-down refinement and makes use of window inference. We then specify the more flexible derivation editor based on refinement diagram notions. We also describe some extensions to the specification that allow for management of how derivations are viewed by the user. Furthermore, we give definitions for syntax- and type-checking as well as for pretty-printing and parsing. Finally, the editor is extended with operations for editing and building expressions.

A note on Z style: We sometimes treat sets as if they were boolean functions, so, for $x : X$, $p : \mathbb{P} X$, we write $p(x)$ for $x \in p$. For relation $R : X \leftrightarrow Y$, we write xRy for $(x, y) \in R$. We omit universal quantifications from axiomatic definitions of sets and functions as the types of the quantified variables are obvious from the type declaration. Finally, in the declaration part of operation schemas, we write $\Delta S(\tilde{x})$, where S is a state schema, and $\tilde{x}$ is a list of state variables in S; here, $\tilde{x}$ acts as a frame for the operation, saying that only those variables in $\tilde{x}$ are allowed to change, while variables of S not in $\tilde{x}$ remain unchanged. A selected Z glossary can be found in Appendix A.

2 Expressions

In this section, we introduce some definitions associated with expressions. We define expressions and paths to subexpressions. We also define substitution and replacement of subexpressions, pattern-matching for expressions, and a notion of equivalence between expressions.

An expression will be modelled as a tree-like structure consisting of operators and expression variables, so we introduce the types *Operator* and *ExVar*:

$$[Operator, ExVar]$$

An expression may be empty, or may be an expression variable, or may consist of an operator and a sequence of operand expressions:

$$Exp ::= empty \mid evar\langle\langle ExVar\rangle\rangle \mid op\langle\langle Operator \times \operatorname{seq} Exp\rangle\rangle$$

Expression variables will be important for pattern-matching and expression substitution as we shall see later. Note that expressions are abstract representations rather than concrete representations consisting of sequences of characters. For example, the arithmetic expression

$3 + 5$ would be represented by the following expression tree:

$$op(+,\ \langle\ op(3, \langle\rangle),\ op(5, \langle\rangle)\ \rangle\ )$$

For an expression tree of the form $op(m, args)$, each $args(i)$ is itself an expression tree. Any nested subtree of an expression tree is reached by following a *path* of argument indices, where a path is a sequence of positive natural numbers:

$$Path == \operatorname{seq} \mathbb{N}_1$$

Given an expression tree e and a path of indices p, the subexpression of e reached by following p is given by $subexp(e, p)$ where the function $subexp$ is defined as follows:

$$\begin{array}{|l}
subexp : (Exp \times Path) \rightarrow\!\!\!\!\!+ Exp \\
\hline
subexp(e, \langle\rangle) = e \\
subexp(op(m, args), \langle i \rangle \frown p) = subexp(args(i), p)
\end{array}$$

The set of paths of an expression tree is given by the following function:

$$\begin{array}{|l}
paths : Exp \rightarrow \mathbb{P}\, Path \\
\hline
paths(empty) = \{\langle\rangle\} \\
paths(evar(v)) = \{\langle\rangle\} \\
paths(op(m, args)) = \\
\qquad \{\langle\rangle\} \cup \{\ i : \mathbb{N}_1;\ \ p : Path \mid i \leq \#args \wedge p \in paths(args(i)) \bullet \langle i \rangle \frown p\ \}
\end{array}$$

Note that $paths(e)$ is well-defined for any $e \in Exp$, i.e., $paths$ is total. However, we only know that $subexp(e, p)$ is well-defined if p is a valid path in e:

$$p : Path;\ \ e : Exp\ \ \vdash\ \ p \in paths(e)\ \Rightarrow\ (e, p) \in \operatorname{dom} subexp$$

The set of expression variables in expression e is given by $evars(e)$:

$$\begin{array}{|l}
evars : Exp \rightarrow \mathbb{P}\, ExVar \\
\hline
evars(evar(v)) = \{v\} \\
evars(op(m, args)) = (\bigcup i : \mathbb{N}_1 \mid i \leq \#args \bullet evars(args(i))\ )
\end{array}$$

A binding is a function that maps expression variables to expressions:

$$Binding == ExVar \rightarrow\!\!\!\!\!+ Exp$$

Given an expression e, and a binding b, the result of substituting all expression variables of e with what they are bound to in b is given by $substitute(e, b)$:

$$\begin{array}{|l}
substitute : (Exp \times Binding) \rightarrow Exp \\
\hline
substitute(empty, b) = empty \\
v \in \operatorname{dom} b\ \Rightarrow\ substitute(evar(v), b) = b(v) \\
v \notin \operatorname{dom} b\ \Rightarrow\ substitute(evar(v), b) = evar(v) \\
substitute(op(m, args), b) = \\
\qquad op(m, \{\ i : \mathbb{N}_1 \mid i \leq \#args \bullet i \mapsto substitute(args(i), b)\ \})
\end{array}$$

Given expressions e, e', and expression path p, the result of replacing the subexpression of e pointed to by p with e' is given by $replace(e, p, e')$:

$$
\begin{array}{|l}
replace : (Exp \times Path \times Exp) \nrightarrow Exp \\
\hline
replace(e, \langle\rangle, e') = e' \\
replace(op(m, args), \langle i \rangle \frown p, e') = \\
\qquad op(m, args \oplus \{\ i \mapsto replace(args(i), p, e')\ \})
\end{array}
$$

$replace(e, p, e')$ is well-defined if p is a valid path of e:

$$p : Path; \quad e, e' : Exp \ \vdash \ p \in paths(e) \Rightarrow (e, p, e') \in \operatorname{dom} replace$$

2.1 Expression Equivalence

Some properties of operators (e.g. commutativity and associativity) are applied quite frequently in the manipulations on expressions. Usually people make use of them almost unconsciously. While preserving formality, we would like to protect users of the editor from the necessity of applying such transformations explicitly. So an equivalence relation between expressions is defined. We assume that the editor will be able to recognise equivalent expressions and to treat them as if they are the same. Here we describe an equivalence relation expressing the commutativity and associativity properties of operators. These could be extended to include other equivalences as well.

Sets *commutative* and *associative* are introduced to represent commutative and associative operators respectively:

$$
\begin{array}{|l}
commutative : \mathbb{P}\, Operator \\
associative : \mathbb{P}\, Operator
\end{array}
$$

The equivalence relation is defined by inference rules where assumptions are represented as a predicate on the left hand side of the implication. Because of the semantics of the implication there exists many relations which satisfy the defining predicate. To have an unambigous definition one of them must be distinguished. We are interested in the smallest relation, the one which has only the properties we have postulated. So a new notation is introduced to express this:

$$least(R) \bullet Q(R).$$

It is equivalent to:

$$Q(R) \land (\forall P : T \mid Q(P) \bullet R \subseteq P),$$

where R has type T and Q is the predicate following in the definition, which specifies relation R.

The equivalence relation is then specified as follows:

$$
\begin{array}{|l}
\_ \overset{E}{\approx} \_ : Exp \leftrightarrow Exp \\
\hline
least(\overset{E}{\approx}) \bullet \\
reflexive(\overset{E}{\approx}) \\
transitive(\overset{E}{\approx}) \\
symmetric(\overset{E}{\approx}) \\
\#as = \#as' \;\wedge\; (\forall\, i \bullet as(i) \overset{E}{\approx} as'(i)) \quad\Rightarrow\quad op(m, as) \overset{E}{\approx} op(m, as') \\
commutative(m) \quad\Rightarrow\quad op(m, \langle e, e' \rangle) \overset{E}{\approx} op(m, \langle e', e \rangle) \\
associative(m) \quad\Rightarrow\quad op(m, \langle e, op(m, \langle e', e'' \rangle) \rangle) \overset{E}{\approx} op(m, \langle op(m, \langle e, e' \rangle), e'' \rangle)
\end{array}
$$

2.2 Pattern Matching

We regard a pattern simply as an expression containing some expression variables and say that a pattern p matches an expression e if there is some binding b such that,

$$e \quad = \quad substitute(p, b).$$

For example, the pattern '$x + y$' matches the specific expression '$(3+4)+5$' when the expression variables 'x' and 'y' are bound to '$(3 + 4)$' and '5' respectively. For our purposes, we do not need to match one pattern with another pattern, rather we only need to match a pattern with a specific expression. Thus, '$x + 5$' will not match with '$(3 + 4) + y$'. To allow for pattern-matching modulo commutativity and associativity, we simply require equivalence rather than equality between the expression and the substituted pattern, i.e., $e \overset{E}{\approx} substitute(p, b)$. We introduce *match* as follows:

$$
\begin{array}{|l}
match : \mathbb{P}(Exp \times Exp \times Binding) \\
\hline
match(p, e, b) \;\Leftrightarrow\; substitute(p, b) \overset{E}{\approx} e
\end{array}
$$

Thus we have that pattern p matches expression e if

$$(\exists\, b : Binding \bullet match(p, e, b)).$$

3 A Linear Derivation Editor

When transforming an expression, some relation must be preserved between the expression before and after it is transformed. We assume that all such transformation relations are reflexive and transitive, and that they respect the equivalence relation on expressions. Transformation relations are therefore modelled by the type *Relation* defined as follows:

$$
\begin{array}{l}
Relation == \{\; R : Exp \leftrightarrow Exp \mid \; reflexive(R) \wedge transitive(R) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad (\forall\, e, e' : Exp \bullet e \overset{E}{\approx} e' \;\Rightarrow\; e \; R \; e') \;\; \}
\end{array}
$$

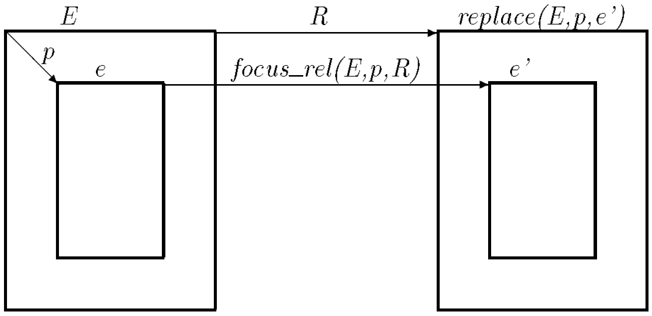

Figure 2: Focus Relation

Furthermore, we assume that there is an ordering on transformation relations such that lesser transformation relations may be used in place of greater transformation relations:

$$\begin{array}{|l}
_ \preceq _ : \mathit{Relation} \leftrightarrow \mathit{Relation} \\
\hline
R \preceq R' \;\wedge\; e\ R\ e' \;\Rightarrow\; e\ R'\ e'
\end{array}$$

For example, in terms of the refinement calculus, we have that program equality may be used in place of program refinement.

The linear editor allows an expression to be manipulated by applying transformations to a subexpression of that expression. The subexpression to which transformations should be applied is called the *focus*. In order to preserve some relation on the main expression, the transformation must preserve some relation on the focus, which need not necessarily be the same relation. We assume the existence of a function *focus_rel* which given an expression E, a relation R to be preserved on E, and a path p to some subexpression of E, returns the relation that should be preserved on the subexpression (see Figure 2):

$$\begin{array}{|l}
\mathit{focus_rel} : (\mathit{Exp} \times \mathit{Path} \times \mathit{Relation}) \nrightarrow \mathit{Relation} \\
\hline
(e, e') \in \mathit{focus_rel}(E, p, R) \;\Rightarrow\; (E, \mathit{replace}(E, p, e')) \in R
\end{array}$$

Here, rather than defining the function *focus_rel*, we have simply assumed that it satisfies a certain property. For example, the window-opening mechanism described in [7] satisfies this property; it calculates the focus relation by selecting an appropriate composition of inference rules of the form described in Section 1; it fails if there are no suitable inference rules (hence *focus_rel* is partial), and if the inference rules allow for more than one candidate focus relation, it selects the greatest one.

A transformation rule consists of a left-hand side pattern which should match with the current focus, a right-hand side pattern which replaces the focus, and a relation that holds between *lhs* and *rhs* for any substitution of the expression variables[1]:

[1] This property of transformation rules which says that the relation must hold under substitution with any binding b is quite strong, and needs to be relaxed in a later version so that side-conditions can be placed on the expressions matching expression variables.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\ TRule\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
lhs, rhs : Exp \\
rel : Relation \\
\hline
\forall\, b : Binding \bullet \\
\quad (substitute(lhs, b), substitute(rhs, b)) \in rel \\
\hline
\end{array}
$$

Each transformation rule used in the editor will be given a unique identifier:

$[TIdent]$

The state variables of the editor are given by the following schema:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\ LinearEditor\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
trules : TIdent \nrightarrow TRule \\
original, main, focus : Exp \\
main\_rel : Relation \\
fptr : Path \\
\hline
fptr \in paths(main) \\
focus = subexp(main, fptr) \\
(original, main) \in main\_rel \\
\hline
\end{array}
$$

The set of available transformation rules is given by *trules*. The variable *main* represents the expression being transformed, *original* preserves the value of *main* that the user starts with, *focus* represents the focus, and *fptr* points to the focus within *main*. The invariant says that *fptr* should always be a valid path of *main*, and specifies how *focus* and *main* are related. The relation to be preserved on the main expression is *main_rel*. For example, if the expressions are programs, then *main_rel* will be the refinement relation. The invariant also specifies that the original expression should be related to the current value of *main* by *main_rel*, i.e., all transformations performed so far should have preserved *main_rel*.

A number of operations for changing the focus are provided. The focus is enlarged by moving up the main expression tree. This is achieved by removing the last index from the focus pointer:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\ Up\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta LinearEditor(fptr, focus) \\
\hline
fptr' = front(fptr) \\
focus' = subexp(main, fptr') \\
\hline
\end{array}
$$

The focus is reduced by moving down to the first argument of the current focus:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\ Down\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta LinearEditor(fptr, focus) \\
\hline
fptr' = fptr \frown \langle 1 \rangle \\
focus' = subexp(main, fptr') \\
\hline
\end{array}
$$

Furthermore, the focus is changed to the expression which is to the right (left) of the current focus by incrementing (decrementing) the last index of the focus pointer:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Right}} \\
\hline
\Delta \mathit{LinearEditor}(\mathit{fptr}, \mathit{focus}) \\
\hline
\mathit{fptr}' = \mathit{front}(\mathit{fptr}) \frown \langle \mathit{last}(\mathit{fptr}) + 1 \rangle \\
\mathit{focus}' = \mathit{subexp}(\mathit{main}, \mathit{fptr}') \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Left}} \\
\hline
\Delta \mathit{LinearEditor}(\mathit{fptr}, \mathit{focus}) \\
\hline
\mathit{fptr}' = \mathit{front}(\mathit{fptr}) \frown \langle \mathit{last}(\mathit{fptr}) - 1 \rangle \\
\mathit{focus}' = \mathit{subexp}(\mathit{main}, \mathit{fptr}') \\
\hline
\end{array}
$$

An operation to explicitly set the current focus is specified as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SetFocus}} \\
\hline
\Delta \mathit{LinearEditor}(\mathit{fptr}, \mathit{focus}) \\
\mathit{fptr}? : \mathit{Path} \\
\hline
\mathit{fptr}' = \mathit{fptr}? \\
\mathit{focus}' = \mathit{subexp}(\mathit{main}, \mathit{fptr}') \\
\hline
\end{array}
$$

The input parameter *fptr*? could be calculated from mouse coordinates, for example.

An expression is transformed by applying a transformation rule selected from the set of available rules. The *lhs* expression of the transformation rule is a pattern that is matched against the current focus. The binding of expression variables that allows them to match is then used to substitute for expression variables in the *rhs* expression, and the focus is replaced by the substituted *rhs*. The *rhs* expression of a transformation rule may contain expression variables that are not in the *lhs*. Bindings for these expression variables must be supplied by the user. For example, when applying a transformation rule for introducing a local variable in a program, the user would have to supply a name for that local variable. The current focus is transformed by the *Transform* operation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Transform}} \\
\Delta LinearEditor(focus, main) \\
i? : TIdent \\
rhsb? : Binding \\
\hline
\exists\, rule : TRule;\ \ b : Binding \bullet \\
\quad rule = trules(i?) \\
\quad \mathrm{dom}\, b = evars(rule.lhs) \\
\quad \mathrm{dom}\, rhsb? = evars(rule.rhs) \setminus evars(rule.lhs) \\
\quad rule.rel \preceq focus\_rel(main, fptr, main\_rel) \\
\quad match(rule.lhs, focus, b) \\
\quad focus' = substitute(rule.rhs, b \cup rhsb?) \\
\quad main' = replace(main, fptr, focus') \\
\hline
\end{array}
$$

The identity of the transformation rule to be applied to the current focus, and the extra binding for *rhs* are given as input parameters. The domain of *rhsb*? should be exactly those expression variables that are extra in *rhs*. The relation preserved by the transformation rule should satisfy the relation given by *focus_rel*. The *lhs* expression should match the focus, and the focus is replaced by *rhs* with expression variables substituted where appropriate.

Calculating the pre-condition of the transformation operation yields:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathrm{pre}\ \textit{Transform}} \\
LinearEditor \\
i? : TIdent \\
rhsb? : Binding \\
\hline
\exists\, b : Binding \bullet \\
match(trules(i?).lhs, focus, b) \\
\mathrm{dom}\, rhsb? = evars(trules(i?).rhs) \setminus evars(trules(i?).lhs) \\
trules(i?).rel \preceq focus\_rel(main, fptr, main\_rel) \\
\hline
\end{array}
$$

So, the *Transform* operation may be validly applied if the *lhs* expression of the rule matches the focus, the *rhsb*? binding binds the expression variables that are extra in the *rhs* of the rule, and the relation preserved by the transformation rule is appropriate. The pre-condition tells us that, under these conditions, *Transform* is guaranteed to preserve the invariant which says that $(original, main) \in main\_rel$. Calculation of the pre-condition of *Transform* is described in full in Appendix B.

4 A Refinement-Diagram Based Editor

The linear editor supports a top-down style of derivation, though as it stands, it doesn't provide facilities for undoing transformations, or for recording the history of transformations that have been applied. In this section, we extend the editor to provide such support. We also extend it in a more general way so that it not only supports top-down derivation, but also allows for the more flexible style that we require. The extended editor will provide facilities for trying out alternative refinements, for moving easily between abstraction levels, and for reusing previous derivations in new contexts.

These extensions are based on the refinement diagram notion [3] which is a graphical description of a sequence of successive transformations of program components, as well as nested derivations. We will model this as a set of transformation relationships between components.

We introduce some new types for the diagram editor. The type *Info* can be some arbitrary (textual) information supplied by the user.

$$[\mathit{Info}]$$

It is used in the type *Status*, which denotes the status of a relation. The status is either *derived* or some text given by the user.

$$\mathit{Status} ::= \mathit{derived} \mid \mathit{given}\langle\!\langle \mathit{Info} \rangle\!\rangle$$

We also introduce an identifier type *Name* that will be used to identify named expressions.

$$[\mathit{Name}]$$

The relations between an expression and its transformed version is stored as a record *TransRel*. It consists of the original expression, the transformation relation and the refined expression, as well as of the status of the relation.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{TransRel}} \\
\hline
\mathit{from}, \mathit{to} : \mathit{Exp} \\
\mathit{rel} : \mathit{Relation} \\
\mathit{stat} : \mathit{Status} \\
\hline
\end{array}$$

The new editor, *Editor*, is a slightly changed instance of our previous editor, *LinearEditor*. Now, *def* is the name of the "original" expression, in the sense that it is the starting point of the current sequence of transformations and *def_rel* is the relation that should be preserved (these "override" the previous *orig* and *main_rel* in *LinearEditor*). The expression *main* is the actual expression we are currently working with. We also add a set of named expressions *exps* and a set of previously derived transformation relations *trels*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Editor}} \\
\hline
def : Name \\
def\_rel : Relation \\
main, focus : Exp \\
fptr : Path \\
trules : TIdent \pfun TRule \\
exps : Name \pfun Exp \\
trels : \mathbb{P}\ TransRel \\
\hline
fptr \in paths(main) \\
focus = subexp(main, fptr) \\
\hline
\end{array}
$$

We describe an operation to identify an arbitrary expression with a name for later reference. The expression could be built from scratch or taken from the focus:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SetName}} \\
\hline
\Delta Editor(exps) \\
newname? : Name \\
exp? : Exp \\
\hline
newname? \notin \operatorname{dom} exps \\
exps' = exps \cup \{newname? \mapsto exp?\} \\
\hline
\end{array}
$$

When we are performing transformations on an expression we first have to choose a starting point. This is done by choosing a named expression and a working relation as in *ChooseMain*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ChooseMain}} \\
\hline
\Delta Editor(def, def\_rel, main, focus, fptr) \\
name? : Name \\
defrel? : Relation \\
\hline
def' = name? \\
def\_rel' = defrel? \\
main' = exps(name?) \\
focus' = main' \\
fptr' = \langle\rangle \\
\hline
\end{array}
$$

We specify a new version of the *Transform* operation which transforms the focus and adds a new *TransRel* to the set *trels* of relations. Since this operation is based on the editors transformation rules, the status of the relation is considered to be ‘derived’:

$$
\begin{array}{|l}
\hline
\textit{Transform} \\
\Delta Editor(focus, main, trels) \\
i? : TIdent \\
rhsb? : Binding \\
\hline
\exists\, rule : TRule;\ \ b : Binding \bullet \\
\quad rule = trules(i?) \\
\quad \operatorname{dom} b = evars(rule.lhs) \\
\quad \operatorname{dom} rhsb? = evars(rule.rhs) \setminus evars(rule.lhs) \\
\quad rule.rel \preceq focus\_rel(main, fptr, def\_rel) \\
\quad match(rule.lhs, focus, b) \\
\quad focus' = substitute(rule.rhs, b \cup rhsb?) \\
\quad main' = replace(main, fptr, focus') \\
\quad trels' = trels \cup \{mk\_TransRel(focus, focus', rule.rel, derived)\} \\
\hline
\end{array}
$$

Here, we assume that $mk\_TransRel$ converts a tuple into an appropriate schema-typed element, i.e., has type:

$$mk\_TransRel : (Exp \times Exp \times Rel \times Status) \rightarrowtail TransRel.$$

The user is also allowed to add new relations to $trels$ (e.g., conjecture relations for proof later). In this case, the relation will have the status $given$:

$$
\begin{array}{|l}
\hline
\textit{GiveRelation} \\
\Delta Editor(trels) \\
e1?, e2? : Name \\
rel? : Relation \\
info? : Info \\
\hline
trels' = trels \cup \{mk\_TransRel(exps(e1?), exps(e2?), rel?, given(info?))\} \\
\hline
\end{array}
$$

4.1 Browsing refinement diagrams

As well as applying a transformation rule to the current focus, we may wish to replace it with previously derived transformations. We regard this as a form of refinement-diagram browsing since it allows for the traversal of a sequence of known transformations of some subcomponent. In terms of our model, we simply have to select a record from $trels$ whose 'from' component corresponds to the current focus and whose 'rel' component is appropriate, and replace it with the 'to' component of that record; thus the operation to replace the focus with a next transformation of it is specified as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NextTrans}} \\
\Delta \textit{Editor}(\textit{main}, \textit{focus}) \\
\hline
\exists\, r : \textit{TransRel} \mid r \in \textit{trels} \bullet \\
\quad \textit{focus} \overset{E}{\approx} r.\textit{from} \\
\quad r.\textit{rel} \preceq \textit{focus\_rel}(\textit{main}, \textit{fptr}, \textit{def\_rel}) \\
\quad \textit{focus}' \overset{E}{\approx} r.\textit{to} \\
\quad \textit{main}' = \textit{replace}(\textit{main}, \textit{fptr}, \textit{focus}') \\
\hline
\end{array}
$$

Notice that we have used expression equivalence rather than equality, allowing the editor to make implicit use of commutativity and associativity. Since a refinement diagram may be a tree-like structure, with a component having many alternative transformations, and not just a linear structure, there may be more than one r in *trels* to choose from. We have left this selection nondeterministic. In a more refined version, the user could be asked to select from a set of alternatives, or we could impose an ordering on the alternatives, and provide operations for moving through them.

The operation for browsing in reverse is specified as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PrevTrans}} \\
\Delta \textit{Editor}(\textit{main}, \textit{focus}) \\
\hline
\exists\, r : \textit{TransRel} \mid r \in \textit{trels} \bullet \\
\quad \textit{focus} \overset{E}{\approx} r.\textit{to} \\
\quad r.\textit{rel} \preceq \textit{focus\_rel}(\textit{main}, \textit{fptr}, \textit{def\_rel}) \\
\quad \textit{focus}' \overset{E}{\approx} r.\textit{from} \\
\quad \textit{main}' = \textit{replace}(\textit{main}, \textit{fptr}, \textit{focus}') \\
\hline
\end{array}
$$

Note that the difference between the *Transform* operation and the browsing operations is that the former requires unification of expression variables with subexpressions, whereas the latter case simply requires pattern matching.

4.2 Derivation consolidation

Having started with some expression (using *ChooseMain*) and having applied some sequence of transformation and browsing operations to it, the user arrives at a new expression. We offer the user the facility to explicitly establish the relationship between what he or she started with (i.e., the *def* component) and the new expression. We refer to this as derivation consolidation.

For example, in Figure 3, the user would have started with expression $m1$ and arrived at expression $m2$. We consolidate this sequence of derivations by recording in *trels* that $m1$ is related to $m2$ by *rel*. If the user had only applied a sequence of *Transform* and *NextRef* operations, then, as with the linear editor, the relationship would automatically hold. However, by allowing

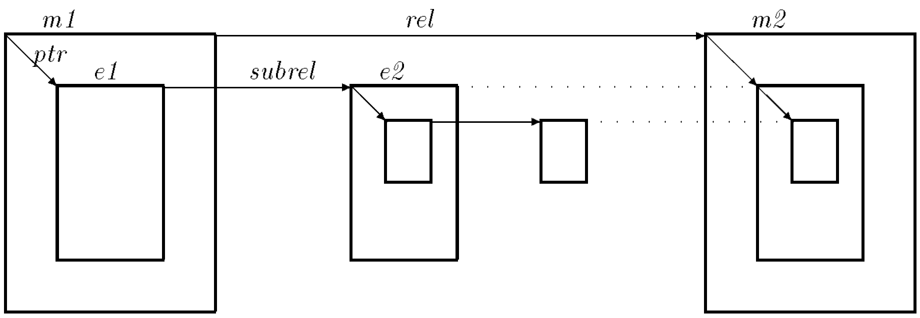

Figure 3: Derivation consolidation

the user to perform *PrevRef* operations, this is no longer guaranteed, so we introduce an explicit test to check whether *rel* holds between $m1$ and $m2$. Thus, $is\_rel\_conn(trels, rel, m1, m2)$ holds if there is a transitive sequence of transformation relations in *trels* that together establish *rel* between $m1$ and $m2$. This auxiliary test is specified in Appendix C. This appendix also describes an auxiliary function $getstat(trels, rel, m1, m2)$ that returns the status of the derivation of $m2$ from $m1$; it has the status 'derived' if all the individual transformations that establish *rel* between $m1$ and $m2$ are derived, otherwise it has the status 'given'.

These auxiliary definitions are used to describe an operation that consolidates the relation from *def* to *main* as follows:

ConsolidateDefToMain

$$\Delta Editor(trels, exps)$$
$$cname? : Name$$

$$cname? \notin \operatorname{dom} exps$$
$$is\_rel\_conn(trels, def\_rel, exps(def), main)$$
$$trels' = trels \cup \{mk\_TransRel(exps(def), main, def\_rel, getstat(trels, def\_rel, exps(def), main))\}$$
$$exps' = exps \cup \{cname? \mapsto main\}$$

If the user has browsed from *def* in reverse, then it may be possible to consolidate the reverse transformation, i.e., establish the relationship from *main* to *def*:

ConsolidateMainToDef

$$\Delta Editor(trels, exps)$$
$$cname? : Name$$

$$cname? \notin \operatorname{dom} exps$$
$$is\_rel\_conn(trels, def\_rel, main, exps(def))$$
$$trels' = trels \cup \{mk\_TransRel(main, exps(def), def\_rel, getstat(trels, def\_rel, main, exps(def)))\}$$
$$exps' = exps \cup \{cname? \mapsto main\}$$

Consolidated relations that have status 'derived' could be introduced in the editors database of transformation rules.

5 View Management

Here we describe some extensions to the editor that provide the user with better control over how expressions are viewed.

5.1 Folding and unfolding expressions

The ability to fold and unfold subexpressions is highly desirable when dealing with large expressions. In order to introduce these the following approach is considered.

Firstly we introduce a type *Flag* which says whether a subexpression is visible or not

$$\mathit{Flag} ::= \mathit{visible} \mid \mathit{invisible}$$

Secondly we change the expression structure slightly by adding a *Flag* to each operator in the following way

$$\mathit{Exp} ::= \mathit{empty} \;\mid\; \mathit{evar}\langle\langle \mathit{ExVar} \rangle\rangle \;\mid\; \mathit{op}\langle\langle \mathit{Operator} \times \mathrm{seq}\, \mathit{Exp} \times \mathit{Flag} \rangle\rangle$$

Folding and unfolding of (sub)expressions is then simply done by toggling this flag in the focus between the states *visible* and *invisible*:

$$\begin{array}{|l}
\mathit{toggle} : \mathit{Exp} \to \mathit{Exp} \\
\hline
\mathit{toggle}(\mathit{empty}) = \mathit{empty} \\
\mathit{toggle}(\mathit{evar}(v)) = \mathit{evar}(v) \\
\mathit{toggle}(\mathit{op}(m, \mathit{args}, \mathit{visible})) = \mathit{op}(m, \mathit{args}, \mathit{invisible}) \\
\mathit{toggle}(\mathit{op}(m, \mathit{args}, \mathit{invisible})) = \mathit{op}(m, \mathit{args}, \mathit{visible})
\end{array}$$

$$\begin{array}{|l|}
\hline
\mathit{Fold/Unfold} \\
\Delta \mathit{Editor}(\mathit{main}, \mathit{focus}) \\
\hline
\mathit{focus}' = \mathit{toggle}(\mathit{focus}) \\
\mathit{main}' = \mathit{replace}(\mathit{main}, \mathit{fptr}, \mathit{focus}') \\
\hline
\end{array}$$

Naturally, the pretty printer will also have to be changed so that when it encounters an operator that is marked *invisible* it, instead of the operator's token, outputs a special token like "..." and stops traversing the current branch of the expression.

5.2 Several views

Another useful feature is the possibility of having several different views of the current expression *main*, where we assume that each view is associated with its own window in the user interface. Each view is given by an offset path (*vptr*) from *main*, and contains its own focus:

$$
\begin{array}{|l}
\hline
\textit{View} \\
\hline
view, focus : Exp \\
vptr, fptr : Path \\
\hline
fptr \in paths(view) \\
subexp(view, fptr) = focus \\
\hline
\end{array}
$$

The editor is extended with a set of views (including a current view) as follows. Each view is identified by a unique identifier.

$[Vid]$

$$
\begin{array}{|l}
\hline
\textit{EditorWithViews} \\
\hline
Editor \\
cv : Vid \\
views : Vid \rightarrowtail\!\!\!\!\!\!+ View \\
\hline
cv \in \mathrm{dom}\, views \\
focus = views(cv).focus \\
fptr = views(cv).vptr \frown views(cv).fptr \\
(\forall v : Vid \mid v \in \mathrm{dom}\, views \bullet \\
views(v).vptr \in paths(main) \Rightarrow \\
\;\; subexp(main, views(v).vptr) = views(v).view) \\
\hline
\end{array}
$$

Changing of the current view is done at the level of the user interface and usually carried out implicitly (by moving the mouse, for example). The effect is specified as follows:

$$
\begin{array}{|l}
\hline
\textit{SetView} \\
\hline
\Delta EditorWithViews(cv, fptr, focus) \\
v? : Vid \\
\hline
v? \in \mathrm{dom}\, views \\
cv' = v? \\
fptr' = views(v?).vptr \frown views(v?).fptr \\
focus' = views(v?).focus \\
\hline
\end{array}
$$

Note The implication in the invariant of *EditorWithViews* should really be an equivalence, and informally it should also be considered as one. However, some operation might invalidate one or more views as a side effect of *main* being changed, and thus making the antecedent of the implication false. We propose that when this happens, the implementation should take one of the following actions. Either a message should be printed in the view concerned, stating that the branch no longer exists, or the view should be closed altogether. In both cases the invalidated view(s) should be removed from *views*.

5.3 Opening and closing views

New views are opened as follows:

$$
\begin{array}{|l}
\hline
\textit{OpenView} \\
\Delta \textit{EditorWithViews}(cv, views) \\
v? : Vid \\
\hline
v? \notin \operatorname{dom} views \\
\exists\, vr' : View \;\bullet \\
vr'.vptr = fptr \\
vr'.view = focus \\
vr'.fptr = \langle\rangle \\
vr'.focus = focus \\
cv' = v? \\
views' = views \cup \{v? \mapsto vr'\} \\
\hline
\end{array}
$$

The closing of a view refers to the current view, after which some other view is picked as the new current view:

$$
\begin{array}{|l}
\hline
\textit{CloseCurrentView} \\
\Delta \textit{EditorWithViews}(fptr, focus, cv, views) \\
\hline
\exists\, v : Vid \mid v \in \operatorname{dom} views \;\bullet \\
v \neq cv \\
cv' = v \\
fptr' = views(v).vptr \frown views(v).fptr \\
focus' = views(v).focus \\
views' = \{cv\} \lhd views \\
\hline
\end{array}
$$

5.4 Shrinking and expanding views

The context shown around the selected expression, *focus*, in a view may be expanded or shrunk. The operation *Shrink* shrinks the view on the screen to the focus, thus, making it easier for the user to work with some details of the derivation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{Shrink}\ }} \\
\Delta \mathit{EditorWithViews}(\mathit{views}) \\
\hline
\exists\, vr' : \mathit{View} \;\bullet \\
\quad vr'.vptr = fptr \\
\quad vr'.view = focus \\
\quad vr'.fptr = \langle\rangle \\
\quad vr'.focus = focus \\
\quad views' = (\{cv\} \ntriangleleft views) \cup \{cv \mapsto vr'\} \\
\hline
\end{array}
$$

With the *Expand* operation a view is extended to the main expression:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{Expand}\ }} \\
\Delta \mathit{EditorWithViews}(\mathit{views}) \\
\hline
\exists\, vr' : \mathit{View} \;\bullet \\
\quad vr'.vptr = \langle\rangle \\
\quad vr'.view = main \\
\quad vr'.fptr = fptr \\
\quad vr'.focus = focus \\
\quad views' = (\{cv\} \ntriangleleft views) \cup \{cv \mapsto vr'\} \\
\hline
\end{array}
$$

Alternatively the *Expand* operation could allow the user to specify how many "levels" should be revealed instead of revealing the complete expression in one step.

6 Other Considerations

We have specified some central features of a program derivation editor. Besides these general aspects, we have also been able to specify features of the editor such as:

- Providing syntax and type rules, and checking expressions against these.
- Pretty-printing of expressions in a concrete syntax and parsing from that concrete syntax to the abstract syntax used here.
- Syntax-directed editing of expressions.

These aspects are described below.

6.1 Syntax Checking

Rather than defining the abstract syntax of a specific language, we describe the general form of rules that will be used to define the syntactic well-formedness of expressions. An expression may be associated with some syntactic category where, for example, *Program*, *Boolean-Expression*, *Arithmetic-Expression*, *State-Variable*, etc, are examples of possible syntactic categories. Syntactic categories are represented by the type *Syntax*:

$$[Syntax]$$

We assume the existence of a set of syntax rules describing the abstract syntax of some language as follows:

$$SynStruct == Operator \times \mathrm{seq}\, Syntax$$
$$SynStruct^{+} ::= syn\langle\!\langle Syntax \rangle\!\rangle \mid struct\langle\!\langle SynStruct \rangle\!\rangle$$

$$\mid\ syntax\_rules : Syntax \rightarrow \mathbb{P}\, SynStruct^{+}$$

The function *syntax_rules* models a BNF grammar that maps each syntactic category to a set of possible syntactic structures that expressions of that category may have. For example, the BNF rules:

$$\begin{array}{rcl} Command & ::= & assign(StateVar, ArithExp) \\ & & \mid if(BoolExp, Command, Command) \\ & & \mid \ldots \\ ArithExp & = & StateVar \\ & & \mid add(ArithExp, ArithExp) \\ & & \mid \ldots \\ & \vdots & \end{array}$$

would be modelled by the function

$$\begin{array}{rcl} \{\ \ Command & \mapsto & \{\ struct(assign, \langle StateVar, ArithExp \rangle), \\ & & \quad struct(if, \langle BoolExp, Command, Command \rangle), \ldots\ \}, \\ ArithExp & \mapsto & \{\ syn(StateVar), \\ & & \quad struct(add, \langle ArithExp, ArithExp \rangle), \ldots\ \}, \ldots\ \ \} \end{array}$$

Now, given an expression e and a syntactic category s, $well\_formed(s, e)$ determines whether e is a well-formed expression of syntax s:

$$\begin{array}{|l} well\_formed : Syntax \leftrightarrow Exp \\ \hline well\_formed(s, op(m, args)) \Leftrightarrow \\ \quad (\exists\, s' : Syntax \bullet \\ \quad\quad syn(s') \in syntax\_rules(s) \wedge \\ \quad\quad well\_formed(s', op(m, args))) \\ \quad \vee \\ \quad (\exists\, ss : \mathrm{seq}\, Syntax \bullet \\ \quad\quad struct(m, ss) \in syntax\_rules(s) \wedge \\ \quad\quad \#ss = \#args \wedge \\ \quad\quad (\forall\, i : \mathbb{N}_1 \mid i \leq \#args \bullet well\_formed(ss(i), args(i)))) \end{array}$$

6.2 Type Checking

Here we describe some rules for defining the type of an expression. We shall assume that types may be some set of given type names or constructed using two specific type constructors, namely function and product constructors. Note that these constructors where chosen for illustrative purposes and the definitions can easily be extended to include other type constructors. The type names are given by:

$$[TyName]$$

An important feature will be the ability to describe polymorphic typing, so type variables are also needed:

$$[TyVar]$$

Now a type can be a type variable, a given type, or a constructed function or product type:

$$\begin{aligned} Type ::= {} & tvar\langle\langle TyVar\rangle\rangle \mid given\langle\langle TyName\rangle\rangle \mid \\ & fun\langle\langle Type \times Type\rangle\rangle \mid prod\langle\langle Type \times Type\rangle\rangle \end{aligned}$$

We say that t is a polymorphic type if it is a type variable or is constructed from one or more type variables.

A type binding maps type variables to types:

$$TyBinding == TyVar \rightarrow\!\!\!\!\!+ \; Type$$

Substitution in types is similar to substitution in expressions:

$$ty\_substitute : (Type \times TyBinding) \rightarrow Type$$

$$\begin{aligned} & v \in \operatorname{dom} b \;\Rightarrow\; ty\_substitute(tvar(v), b) = b(v) \\ & v \notin \operatorname{dom} b \;\Rightarrow\; ty\_substitute(tvar(v), b) = tvar(v) \\[6pt] & ty\_substitute(given(g), b) = given(g) \\ & ty\_substitute(fun(t, t'), b) = \\ & \qquad fun(ty\_substitute(t, b), ty\_substitute(t', b)) \\ & ty\_substitute(prod(t, t'), b) = \\ & \qquad prod(ty\_substitute(t, b), ty\_substitute(t', b)) \end{aligned}$$

Pattern matching is used to determine whether some type t is an instance of a polymorphic type p ($instance(p, t)$):

$$instance : Type \leftrightarrow Type$$

$$instance(p, t) \Leftrightarrow (\exists\, b : TyBinding \bullet ty\_substitute(p, b) = t)$$

To allow for currying of functions with product arguments, and to allow for associativity of products we define the following equivalence on types:

$$
\begin{array}{|l}
\_ \stackrel{T}{\approx} \_ : Type \leftrightarrow Type \\
\hline
least(\stackrel{T}{\approx}) \bullet \\[1ex]
reflexive(\stackrel{T}{\approx}) \\
transitive(\stackrel{T}{\approx}) \\
symmetric(\stackrel{T}{\approx}) \\[1ex]
fun(prod(t, t'), t'') \stackrel{T}{\approx} fun(t, fun(t', t'')) \\
prod(prod(t, t'), t'') \stackrel{T}{\approx} prod(t, prod(t', t'')) \\[1ex]
t \stackrel{T}{\approx} t'' \ \wedge \ t' \stackrel{T}{\approx} t''' \ \Rightarrow \\
\qquad prod(t, t') \stackrel{T}{\approx} prod(t'', t''') \wedge \\
\qquad fun(t, t') \stackrel{T}{\approx} fun(t'', t''')
\end{array}
$$

Given a non-empty sequence of types, the function $prod^*$ returns a nested product type that associates to the right:

$$
\begin{array}{|l}
prod^* : \mathrm{seq}_1\ Type \rightarrow Type \\
\hline
prod^*(\langle t \rangle) = t \\
prod^*(\langle t \rangle \frown ts) = prod(t, prod^*(ts))
\end{array}
$$

A signature associates some (possibly) polymorphic type with each operator:

$$
Signature == Operator \rightarrow\!\!\!\!\!\!+\ Type
$$

A type rule $(m, \langle t_1 \ldots t_n \rangle, r)$ associates an operator m with a sequence of operand types $t_1 \ldots t_n$ and a resulting type r. Such a rule says that if m has arguments $a_1 \ldots a_n$ of type $t_1 \ldots t_n$ respectively, then $op(m, \langle a_1 \ldots a_n \rangle)$ is of type r. Given a type signature the function $type\_rule$ returns a set of such type rules:

$$
\begin{array}{|l}
type\_rule : Signature \rightarrow \mathbb{P}(Operator \times \mathrm{seq}\ Type \times Type) \\
\hline
(m, \langle\rangle, t) \in type\_rule(\Sigma) \Leftrightarrow \\
\qquad (\exists\, t' : Type \bullet instance(\Sigma(m), t') \wedge t' \stackrel{T}{\approx} t) \\[1ex]
(m, ts, t) \in type\_rule(\Sigma) \Leftrightarrow \\
\qquad (\exists\, t' : Type \bullet instance(\Sigma(m), t') \wedge t' \stackrel{T}{\approx} fun(prod^*(ts), t))
\end{array}
$$

Here, the type rules associated with an operator m are derived from all types that are equivalent instantiations of $\Sigma(m)$.

Now given a signature Σ, $well\_typed(\Sigma)$ relates types to all expressions of that type:

$$
\begin{array}{|l}
\mathit{well\_typed} : \mathit{Signature} \nrightarrow (\mathit{Type} \leftrightarrow \mathit{Exp}) \\
\hline
(t, op(m, args)) \in \mathit{well\_typed}(\Sigma) \Leftrightarrow \\
\quad (\exists\, ts : \mathrm{seq}\ \mathit{Type} \bullet \\
\quad\quad (m, ts, t) \in \mathit{type\_rule}(\Sigma) \wedge \\
\quad\quad \#ts = \#args \wedge \\
\quad\quad (\forall\, i : \mathbb{N}_1 \mid i \leq \#args \bullet (ts(i), args(i)) \in \mathit{well\_typed}(\Sigma)))
\end{array}
$$

Thus, e is an expression of type t under signature Σ if $(t, e) \in \mathit{well\_typed}(\Sigma)$.

6.3 Pretty Printing and Parsing

Pretty printing of expressions involves converting abstract expression trees into concrete representations of expressions. A concrete expression is modelled as a sequence of tokens (possibly including, for example, line-break tokens):

$$
\begin{array}{l}
[\mathit{Token}] \\
\mathit{CExp} == \mathrm{seq}\ \mathit{Token}
\end{array}
$$

We shall associate pretty-printing rules with operators where a pretty-printing rule consists of a sequence of tokens and argument placeholders:

$$\mathit{PrettySeq} == \mathrm{seq}(\ tok\langle\langle \mathit{Token} \rangle\rangle \mid arg\langle\langle \mathbb{N} \rangle\rangle\ )$$

For example, the pretty-printing rule for an expression formed from the sequential composition operator may look like

$$\langle\ arg(1),\ tok(;\ ),\ arg(2)\ \rangle,$$

meaning first pretty-print the 1st agrument, then print a ";" token, then pretty-print the 2nd agrument. The pretty-printing rule for an **if...then...else...fi** expression may look like:

$$\langle\ tok(\mathbf{if}),\ arg(1),\ tok(\mathbf{then}),\ arg(2),\ tok(\mathbf{else}),\ arg(3),\ tok(\mathbf{fi})\ \rangle.$$

We shall allow for the possiblity of an operator being associated with more than one pretty-printing rule. This would allow for different layouts of an expression depending on, for example, the size or the number of operands. Also, each expression variable is associated with a token. So we assume the following values:

$$
\begin{array}{|l}
\mathit{pretty\_op} : \mathit{Operator} \leftrightarrow \mathit{PrettySeq} \\
\mathit{pretty\_var} : \mathit{ExVar} \rightarrow \mathit{Token}
\end{array}
$$

The relation *pretty* then relates an abstract expression to all possible concrete representations of that expression:

$$
\begin{array}{|l}
pretty : Exp \leftrightarrow CExp \\
pretty_1 : \mathbb{P}((\mathrm{seq}\, Exp) \times PrettySeq \times CExp) \\
\hline
pretty(empty, ts) \Leftrightarrow (ts = \langle\rangle) \\
pretty(evar(v), \langle t \rangle) \Leftrightarrow (t = pretty\_var(v)) \\
pretty(op(m, args), ts) \Leftrightarrow \\
\qquad (\exists\, ps_1 : PrettySeq \bullet pretty\_op(m, ps_1) \wedge pretty_1(args, ps_1, ts) \\
\\
pretty_1(args, \langle\rangle, \langle\rangle) \\
pretty_1(args, \langle tok(t) \rangle \frown ps, ts) \Leftrightarrow \\
\qquad ts \neq \langle\rangle \wedge (head\ ts = t) \wedge pretty_1(args, ps, tail\ ts) \\
pretty_1(args, \langle arg(n) \rangle \frown ps, ts) \Leftrightarrow \\
\qquad (\exists\, ts_1, ts_2 : CExp \mid ts_1 \frown ts_2 = ts \bullet \\
\qquad\qquad pretty(args(n), ts_1) \wedge pretty_1(args, ps, ts_2))
\end{array}
$$

We leave open the criteria for choosing which concrete expression the editor should display.

As well as allowing for the same abstract expression to be related to several concrete expressions, our specification allows the same concrete expression to be related to more than one abstract expression. This is unsatisfactory since it may make it ambiguous what a concrete expression signifies. However, rather than eliminating such ambiguity completely, we simply stipulate that when the same concrete expression corresponds to two different abstract expressions, then those two abstract expressions are equivalent (under commutativity and associativity):

$$
\begin{array}{|l}
pretty(e_1, c) \ \wedge \ pretty(e_2, c) \quad \Rightarrow \quad e_1 \stackrel{E}{\approx} e_2
\end{array}
$$

This restriction must be taken into account when designing the pretty-printing rules.

Parsing consists of converting a concrete expression into an abstract expression, and so we define *parse* to be the inverse of *pretty*:

$$
\begin{array}{|l}
parse : CExp \leftrightarrow Exp \\
\hline
parse = pretty^{\sim}
\end{array}
$$

Because of the above restriction controlling the ambiguity of *pretty*, *parse* will only relate a concrete expression to abstract expressions that are equivalent to each other.

6.4 Syntax Directed Editing

We assume that expressions we are interested in will be constructed by applying a set of transformation rules. However an expression to start from must be built in another way. So we specify basic operations for building a new expression or editing an existing one.

A new (empty) expression is created by the operation *NewExpression*:

$$
\begin{array}{|l}
\hline \textit{NewExpression} \\
\Delta \textit{Editor}(\textit{main}, \textit{focus}, \textit{fptr}) \\
\hline
\textit{main}' = \textit{empty} \\
\textit{fptr}' = \langle\rangle \\
\textit{focus}' = \textit{empty} \\
\hline
\end{array}
$$

Now we define two basic operations for building expressions. The first operation *AddExpression* them corresponds to editing in the syntax-directed fashion. Hence, we introduce a type for identifiers of syntactic structures:

$[\textit{SIdent}]$

We also assume existence of a function which maps an identifier into a corresponding syntactic structure:

$\textit{synstruct} : \textit{SIdent} \nrightarrow \textit{SynStruct}$

The operation *AddExpression* then supplied with an identifier replaces the focus (provided it is empty) by a new expression built from the corresponding operator and a sequence of empty arguments. The length of this sequence can vary for an operator depending on the particular syntactic structure chosen.

$$
\begin{array}{|l}
\hline \textit{AddExpression} \\
\Delta \textit{Editor}(\textit{main}, \textit{focus}, \textit{fptr}) \\
\textit{id}? : \textit{SIdent} \\
\hline
\exists\, m : \textit{Operator};\ \ \textit{ss} : \mathrm{seq}\ \textit{SynStruct} \bullet \\
m = \textit{fst}(\textit{synstruct}(\textit{id}?)) \\
\textit{ss} = \textit{snd}(\textit{synstruct}(\textit{id}?)) \\
\textit{focus} = \textit{empty} \\
\textit{main}' = \textit{replace}(\textit{main}, \textit{fptr}, \textit{op}(m, \langle i : \mathbb{N}_1 \mid i \leq \#\textit{ss} \bullet \textit{empty}\rangle)) \\
\textit{Down} \\
\hline
\end{array}
$$

The second operation *EnterText* allows normal text editing. Taking as an argument an expression in the concrete syntax it parses the latter and replaces the focus (provided it is empty) by output of the parser:

$$
\begin{array}{|l}
\hline \textit{EnterText} \\
\Delta \textit{Editor}(\textit{main}, \textit{focus}) \\
\textit{cexp}? : \textit{CExp} \\
\hline
\exists\, e : \textit{Exp} \bullet \\
\textit{parse}(\textit{cexp}?, e) \\
\textit{focus} = \textit{empty} \\
\textit{main}' = \textit{replace}(\textit{main}, \textit{fptr}, e) \\
\textit{focus}' = e \\
\hline
\end{array}
$$

To manipulate existing expressions we provide usual editing operations. The schema *Editor* is extended by a new state variable *buffer*. It is a temporal store for expressions being manipulated:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{EditorWithBuffer}} \\ \hline
\textit{Editor} \\
\textit{buffer} : \textit{Exp} \\ \hline
\end{array}$$

Operation *Cut* moves an expression in the focus into the buffer, and *CopyToClipped* just copies it:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Cut}} \\ \hline
\Delta \textit{EditorWithBuffer}(\textit{main}, \textit{focus}, \textit{buffer}) \\ \hline
\textit{buffer}' = \textit{focus} \\
\textit{main}' = \textit{replace}(\textit{main}, \textit{fptr}, \textit{empty}) \\
\textit{focus}' = \textit{empty} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CopyToClipped}} \\ \hline
\Delta \textit{EditorWithBuffer}(\textit{buffer}) \\ \hline
\textit{buffer}' = \textit{focus} \\ \hline
\end{array}$$

CopyFromClipped is just inverse of *CopyToClipped*, while *Paste* swaps contents of the current focus and the buffer:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CopyFromClipped}} \\ \hline
\Delta \textit{EditorWithBuffer}(\textit{main}, \textit{focus}) \\ \hline
\textit{main}' = \textit{replace}(\textit{main}, \textit{fptr}, \textit{buffer}) \\
\textit{focus}' = \textit{buffer} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Paste}} \\ \hline
\Delta \textit{EditorWithBuffer}(\textit{main}, \textit{focus}, \textit{buffer}) \\ \hline
\textit{buffer}' = \textit{focus} \\
\textit{main}' = \textit{replace}(\textit{main}, \textit{fptr}, \textit{buffer}) \\
\textit{focus}' = \textit{buffer} \\ \hline
\end{array}$$

7 Concluding Remarks

We believe that the main contribution of this paper to be the combination of window inference with refinement diagrams in the definition of a program derivation editor. Window inference provides a rigorous way of relating derivations of programs with derivations of program subcomponents, while refinement diagrams provide a flexible way of managing program derivations that reflects the way programmers work in practice. We have also incorporated commutativity and associativity properties of operators into transformations, since such properties should be used automatically in a practical tool. Furthermore, syntax and type checking of expressions are defined as well as pretty-printing and parsing. Our specification has helped to clarify the way in which these ideas combine and it serves as a basis for a correct implementation of the editor.

The Centipede editor [4], previously developed within our research group, incorporates some of the refinement diagram notions. It provides graphical representation of refinement diagrams, as well as textual representations of components. However, the manipulation facilities provided are restricted in that only one level of nesting of subcomponents is allowed, and alternative transformations are not allowed. Neither does it incorporate window inference ideas nor commutativity and associativity.

The idea of incorporating associativity and commutativity into transformations was partly inspired by the Proxac editor of van de Snepscheut et al [15] which provides some automatic support for this. Our model of expressions and pattern-matching in Section 2 are quite similar models developed by by Fitzgerald [5] and Vadera [14] using VDM. Naftalin [10] has described a refinement process that is similar to the refinement diagram idea, though the main emphasis there is on incorporating informal design strategies into the refinement process.

Some of the activities of the derivation editor, such as calculating the focus relation, and discharging side-conditions, require mechanical theorem-proving in order to ensure their correctness. An on-going activity in our research group is the development of a refinement tool based on the HOL theorem prover [17]. Much of the refinement calculus has been mechanised in HOL and combined with Grundy's [7] HOL mechanisation of window inference. A powerful X-Windows based user-interface has been developed and this will incorporate some of the ideas presented in this specification.

Acknowledgements

This work was part of the Academy of Finland funded IRENE project at Åbo Akademi. We wish to thank various members of the IRENE project for much fruitful discussion especially Ralph Back.

A Selected Z Glossary

A.1 Logic

$\wedge, \vee$	logical-and, logical-or
$\neg$	negation
$\Rightarrow$	logical implication
$\Leftrightarrow$	equivalence
$(\exists\, x \bullet P)$	exists an x such that P
$(\forall\, x \bullet P)$	forall x, P holds

A.2 Types

A declaration of the form $x : T$ declares x to be of type T. T may be a basic type such as $\mathbb{N}$ or $\mathbb{N}_1$ (naturals and non-zero naturals respectively). A basic type T may also be introduced by the declaration

$$[T].$$

This allows us to introduce a type T without saying explicitly what it contains.

A composite type is built from other types using type constructors. Examples include:

$\mathbb{P}\, T$	powerset of T
$S \times T$	cartesian product of S and T
$S \leftrightarrow T$	relations from S to T
$S \nrightarrow T$	partial functions from S to T
$S \rightarrow T$	total functions from S to T
$\mathrm{seq}\, T$	sequences of elements of type T.

A recursive type is defined using a declaration of the following form:

$$T ::= c_1 \mid \ldots \mid c_n \mid d_1\langle\!\langle E_1[T]\rangle\!\rangle \mid \ldots \mid d_m\langle\!\langle E_m[T]\rangle\!\rangle.$$

Here $E_1[T] \ldots E_2[T]$ are type expressions possibly containing T. For example, we may define the type *NatList* (list of natural numbers) as follows:

$$NatList ::= nil \mid cons\langle\!\langle \mathbb{N} \times NatList\rangle\!\rangle.$$

We may reason about such types inductively.

A.3 Sets

$\{\cdots\}$	set delimiters
$\in, \notin$	set membership, non-membership
$\cup, \cap$	set union, set intersection
$\setminus$	set minus
$S \subseteq T$	S is a subset of T
$\{\ x : T \mid P \bullet t(x)\ \}$	the set of values $t(x)$ for those x of type T satisfying P
$\{\ x : T \mid P\ \}$	the set of those x of type T satisfying P

A.4 Relations and Functions

$x \mapsto y$	mapping of element x to element y
$f(x)$	application of function f to element x
f^{-1}	inverse of function f
$\mathrm{dom}\, f, \mathrm{ran}\, f$	domain of f, range of f
ID_S	identity function on set S
$f \oplus g$	function f overridden by function g
$S \ntriangleleft f$	function f with all elements in S removed from its domain

A.5 Sequences

$\langle \cdots \rangle$	sequence delimiters
$\#s$	length of sequence s
$s \frown t$	s concatenated with t

A.6 Axiomatic Definitions

A constant c may be declared by an axiomatic definition which has the following form:

$$\begin{array}{|l} c : T \\ \hline P(c) \end{array}$$

Here, c is declared to be an object of type T satisfying predicate P.

For example, the constant function *len*, which gives the length of a *NatList* may be declared as follows:

$$\begin{array}{|l} len : NatList \rightarrow \mathbb{N} \\ \hline \forall\, l : NatList; \quad n : \mathbb{N} \bullet \\ len(nil) = 0 \\ len(cons(n, l)) = 1 + len(l) \end{array}$$

A.7 Schemas

A Z schema consists of a name, variable declarations, and a predicate:

$$\begin{array}{|l|} \hline SchemaName \\ x : X \qquad \text{(variable declarations)} \\ \hline Predicate \qquad \text{(predicate relating variables)} \\ \hline \end{array}$$

We illustrate the schema notation of Z by specifying a simple counter system. The state space for the counter is specified by the schema *Counter*:

$$
\begin{array}{|l}
\hline
\textit{Counter} \\
ctr, max : \mathbb{N} \\
\hline
ctr \leq max \\
\hline
\end{array}
$$

The state space consists of a two components of type natural: ctr, representing the current value of the counter, and max, representing the maximum value that the counter may attain. The predicate part of *Counter* is an invariant constraining the value of the counter never to be greater than max.

An operation to increment the counter is defined by the following schema:

$$
\begin{array}{|l}
\hline
\textit{Increment} \\
\Delta Counter(ctr) \\
\hline
ctr < max \\
ctr' = ctr + 1 \\
\hline
\end{array}
$$

The declaration $\Delta Counter(ctr)$ means that this operation updates the state variables in *Counter*, but only the value of ctr may change, while the value of max remains constant.[2] The new value of the counter is referred to as ctr', while the old value is ctr. In general, the Δ-list is a (possibly empty) list of the state variables that may be changed by the operation. The predicate part of this schema simply states that provided ctr is less than max initially, the new value of ctr is the old value plus 1. (There is an implicit conjunction between successive lines in the predicate part.)

A.8 Schemas as Types

Schemas may be used in as types. Essentially they define record structures, with the variables of a schema representing the fields of a record. For example, we may define a schema $Date$: 1994/11/3017 : 15 : 34 as follows:

$$
\begin{array}{|l}
\hline
\textit{Date} \\
day : Day \\
month : Month \\
year : Year \\
\hline
\end{array}
$$

If $today : Date$ has type $Date$: 1994/11/3017 : 15 : 34, then the separate fields of $today$ are referred to as $today.day$, $today.month$ and $today.year$ respectively.

B Pre-condition Calculation

The pre-condition of *Transform* is as follows:

[2] Note: Our Δ-list convention is slightly different to that used in standard Z.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\text{pre } \mathit{Transform}} \\
\hline
\mathit{Editor} \\
i? : \mathit{TIdent} \\
\mathit{rhsb}? : \mathit{Binding} \\
\hline
\exists \mathit{Editor}' \bullet \\
\quad \mathit{trules}' = \mathit{trules} \\
\quad \mathit{original}' = \mathit{original} \\
\quad \mathit{main\_rel}' = \mathit{main\_rel} \\
\quad \mathit{fptr} = \mathit{fptr}' \\
\quad \mathit{fptr}' \in \mathit{paths}(\mathit{main}') \\
\quad \mathit{focus}' = \mathit{subexp}(\mathit{main}', \mathit{fptr}') \\
\quad (\mathit{original}', \mathit{main}') \in \mathit{main\_rel}' \\
\quad \exists\, b : \mathit{Binding};\ \ \mathit{rule} : \mathit{TRule} \bullet \\
\quad \operatorname{dom} b = \mathit{evars}(\mathit{rule}.\mathit{lhs}) \\
\quad \operatorname{dom} \mathit{rhsb}? = \mathit{evars}(\mathit{rule}.\mathit{rhs}) \setminus \mathit{evars}(\mathit{rule}.\mathit{lhs}) \\
\quad \mathit{rule} = \mathit{trules}(i?) \\
\quad \mathit{rule}.\mathit{rel} \preceq \mathit{focus\_rel}(\mathit{main}, \mathit{fptr}, \mathit{main\_rel}) \\
\quad \mathit{match}(\mathit{rule}.\mathit{lhs}, \mathit{focus}, b) \\
\quad \mathit{focus}' = \mathit{substitute}(\mathit{rule}.\mathit{rhs}, b \cup \mathit{rhsb}?) \\
\quad \mathit{main}' = \mathit{replace}(\mathit{main}, \mathit{fptr}, \mathit{focus}') \\
\hline
\end{array}
$$

By several applications of the 1-point rule, this is simplified to:

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\text{pre } \mathit{Transform}} \\
\hline
\mathit{Editor} & \\
i? : \mathit{TIdent} & \\
\mathit{rhsb}? : \mathit{Binding} & \\
\hline
\exists \mathit{focus}' : \mathit{Exp};\ \ b : \mathit{Binding} \bullet & \\
\mathit{fptr} \in \mathit{paths}(\mathit{replace}(\mathit{main}, \mathit{fptr}, \mathit{focus}')) & (1) \\
\mathit{focus}' = \mathit{subexp}(\mathit{replace}(\mathit{main}, \mathit{fptr}, \mathit{focus}'), \mathit{fptr}) & (2) \\
(\mathit{original}, \mathit{replace}(\mathit{main}, \mathit{fptr}, \mathit{focus}')) \in \mathit{main\_rel} & (3) \\
\operatorname{dom} b = \mathit{evars}(\mathit{trules}(i?).\mathit{lhs}) & (4) \\
\operatorname{dom} \mathit{rhsb}? = \mathit{evars}(\mathit{trules}(i?).\mathit{rhs}) \setminus \mathit{evars}(\mathit{trules}(i?).\mathit{lhs}) & (5) \\
\mathit{trules}(i?).\mathit{rel} \preceq \mathit{focus\_rel}(\mathit{main}, \mathit{fptr}, \mathit{main\_rel}) & (6) \\
\mathit{match}(\mathit{trules}(i?).\mathit{lhs}, \mathit{focus}, b) & (7) \\
\mathit{focus}' = \mathit{substitute}(\mathit{trules}(i?).\mathit{rhs}, b \cup \mathit{rhsb}?) & (8) \\
\hline
\end{array}
$$

The following lemmas will be used to eliminate (1) and (2):

Lemma 1 $\forall\, p : \mathit{Paths};\ \ e, e' : \mathit{Exp} \bullet p \in \mathit{paths}(e) \Rightarrow p \in \mathit{paths}(\mathit{replace}(e, p, e'))$

LEMMA 2 $\forall p : Paths; \quad e, e' : Exp \bullet p \in paths(e) \Rightarrow subexp(replace(e, p, e'), p) = e'$

Now (1) and (2) may be eliminated since:

$$\begin{array}{ll} & fptr \in paths(replace(main, fptr, focus')) \\ \Leftarrow & \qquad \{\text{Lemma 1}\} \\ & fptr \in paths(main) \\ \Leftarrow & \qquad \{\text{invariant of } Editor\} \\ & true. \end{array}$$

$$\begin{array}{ll} & focus' = subexp(replace(main, fptr, focus'), fptr) \\ \Leftarrow & \qquad \{\text{Lemma 2}\} \\ & fptr \in paths(main) \\ \Leftarrow & \qquad \{\text{invariant of } Editor\} \\ & true. \end{array}$$

Next, (3) may be eliminated:

$$\begin{array}{ll} & (original, replace(main, fptr, focus')) \in main\_rel \\ \Leftarrow & \qquad \{\text{invariant of } Editor \text{ and transitivity of } main\_rel\} \\ & (main, replace(main, fptr, focus')) \in main\_rel \\ \Leftarrow & \qquad \{main = replace(main, fptr, focus)\} \\ & (replace(main, fptr, focus), replace(main, fptr, focus')) \in main\_rel \\ \Leftarrow & \qquad \{(6)\} \\ & (focus, focus') \in trules(i?).rel \\ \Leftarrow & \qquad \{(8)\} \\ & (focus, substitute(trules(i?).rhs, b \cup rhsb?)) \in trules(i?).rel \\ \Leftarrow & \qquad \{(7) \;\Rightarrow\; focus \overset{E}{\approx} substitute(trules(i?).lhs, b)\} \\ & (substitute(trules(i?).lhs, b), \\ & \quad substitute(trules(i?).rhs, b \cup rhsb?)) \in trules(i?).rel \\ \Leftarrow & \qquad \{(5)\} \\ & (substitute(trules(i?).lhs, b \cup rhsb?), \\ & \quad substitute(trules(i?).rhs, b \cup rhsb?)) \in trules(i?).rel \\ \Leftarrow & \qquad \{trules(i?) \in TRule\} \\ & b \cup rhsb? \in Binding \\ \Leftarrow & \qquad \{(4) \text{ and } (5)\} \\ & true. \end{array}$$

Eliminating (1), (2), and (3), and applying the 1-point rule to $focus'$ gives

$$
\begin{array}{|l}
\text{pre } \mathit{Transform} \\
\mathit{Editor} \\
i? : \mathit{TIdent} \\
\mathit{rhsb}? : \mathit{Binding} \\
\hline
\exists\, b : \mathit{Binding} \bullet \\
\quad \operatorname{dom} b = \mathit{evars}(\mathit{trules}(i?).\mathit{lhs}) \qquad (4) \\
\quad \operatorname{dom} \mathit{rhsb}? = \mathit{evars}(\mathit{trules}(i?).\mathit{rhs}) \setminus \mathit{evars}(\mathit{trules}(i?).\mathit{lhs}) \qquad (5) \\
\quad \mathit{trules}(i?).\mathit{rel} \preceq \mathit{focus\_rel}(\mathit{main}, \mathit{fptr}, \mathit{main\_rel}) \qquad (6) \\
\quad \mathit{match}(\mathit{trules}(i?).\mathit{lhs}, \mathit{focus}, b) \qquad (7) \\
\hline
\end{array}
$$

Finally, the following lemma allows us to eliminate (4):

LEMMA **3** $\forall\, p, e : \mathit{Exp};\quad b : \mathit{Binding} \bullet$

$$(\exists\, b : \mathit{Binding} \bullet \operatorname{dom} b = \mathit{evars}(e) \wedge \mathit{match}(p, e, b)) \Leftrightarrow (\exists\, b : \mathit{Binding} \bullet \mathit{match}(p, e, b)).$$

Thus, pre *Transform* is simplified to:

$$
\begin{array}{|l}
\text{pre } \mathit{Transform} \\
\mathit{Editor} \\
i? : \mathit{TIdent} \\
\mathit{rhsb}? : \mathit{Binding} \\
\hline
\exists\, b : \mathit{Binding} \bullet \\
\operatorname{dom} \mathit{rhsb}? = \mathit{evars}(\mathit{trules}(i?).\mathit{rhs}) \setminus \mathit{evars}(\mathit{trules}(i?).\mathit{lhs}) \\
\mathit{trules}(i?).\mathit{rel} \preceq \mathit{focus\_rel}(\mathit{main}, \mathit{fptr}, \mathit{main\_rel}) \\
\mathit{match}(\mathit{trules}(i?).\mathit{lhs}, \mathit{focus}, b) \\
\hline
\end{array}
$$

C Consolidation Test

We need to introduce an auxiliary definition that determines whether transformation, or reverse transformation, holds between the original and main expressions. Transformation between two expressions will hold if there is a series of transitive transformation steps between subcomponents of these expressions (see Figure 3). First, consider such an individual transformation step. We require that relation *rel* holds between expressions $m1$ and $m2$. If relation *subrel* holds between expressions $e1$ and $e2$ (both corresponding subexpressions of $m1$ and $m2$ respectively), then $\mathit{rel\_step}(\mathit{subrel}, e1, e2, \mathit{rel}, m1, m2)$ determines whether or not *rel* holds between $m1$ and $m2$:

$$
\begin{array}{|l}
rel\_step : \mathbb{P}\ (Relation \times Exp \times Exp \times Relation \times Exp \times Exp) \\
\hline
rel\_step\ (subrel, e1, e2, rel, m1, m2) \Leftrightarrow \\
\quad (\exists\, ptr : Path \bullet \\
\qquad subexp(m1, ptr) = e1 \\
\qquad replace(m1, ptr, e2) = m2 \\
\qquad subrel \preceq focus\_rel(m1, ptr, rel)\ )
\end{array}
$$

In our case, individual transformation steps are of type *TransRel* and are recorded in *trels*. Given such a set of transformation steps *rcs*, the relation $rel\_conn\_step(rcs, rel)(m1, m2)$ determines whether there is some transformation step in *rcs* that causes $m1$ and $m2$ to be related by *rel*.
$rel\_conn\_derived\_step(rcs, rel)\ (m1, m2)$ is similar except that the transformation step from *rcs* must have the status 'derived':

$$
\begin{array}{|l}
rel\_conn\_step : (\mathbb{P}\ TransRel) \times Relation \rightarrow (Exp \leftrightarrow Exp) \\
rel\_conn\_derived\_step : (\mathbb{P}\ TransRel) \times Relation \rightarrow (Exp \leftrightarrow Exp) \\
\hline
rel\_conn\_step\ (rcs, rel)\ (m1, m2) \Leftrightarrow \\
\quad (\exists\, rc : TransRel \mid rc \in rcs \bullet \\
\qquad rel\_step(rc.rel, rc.from, rc.to, rel, m1, m2)\ ) \\
\\
rel\_conn\_derived\_step(rcs, rel)\ (m1, m2) \Leftrightarrow \\
\quad (\exists\, rc : TransRel \mid rc \in rcs\ \wedge\ rc.status = derived \bullet \\
\qquad rel\_step(rc.rel, rc.from, rc.to, rel, m1, m2)\ )
\end{array}
$$

Note that $rel\_conn\_step(rcs, rel)$ above is a relation between expressions. To determine whether $m1$ and $m2$ are related by *rel* through a series of zero or more steps, we simply take the reflexive, transitive closure of $rel\_conn\_step(rcs, rel)$, hence the definitions of *is_rel_conn* and *is_rel_conn_derived*:

$$
\begin{array}{|l}
is\_rel\_conn : (\mathbb{P}\ TransRel) \times Relation \rightarrow (Exp \leftrightarrow Exp) \\
is\_rel\_conn\_derived : (\mathbb{P}\ TransRel) \times Relation \rightarrow (Exp \leftrightarrow Exp) \\
\hline
is\_rel\_conn(rcs, rel) = (rel\_conn\_step\ (rcs, rel))^* \\
\\
is\_rel\_conn\_derived(rcs, rel) = (rel\_conn\_step\_derived\ (rcs, rel))^*
\end{array}
$$

Now $is\_rel\_conn(rcs, rel)(m1, m2)$ determines whether there is a series of transformation steps from *rcs* that cause $m1$ and $m2$ to be related by *rel*. $is\_rel\_conn\_derived(rcs, rel)(m1, m2)$ is similar, except that each step must have the status 'derived'. If $m1$ and $m2$ are related by *rel*, but not through a series of derived relations, then we say that they are related by a 'given' relation:

$$
\begin{array}{|l}
is\_rel\_conn\_given : (\mathbb{P}\ TransRel) \times Relation \rightarrow (Exp \leftrightarrow Exp) \\
\hline
is\_rel\_conn\_given(rcs, rel)\ (m1, m2) \Leftrightarrow \\
\quad is\_rel\_conn\ (rcs, rel)\ (m1, m2)\ \wedge \\
\quad \neg\ is\_rel\_conn\_derived(rcs, rel)\ (m1, m2)
\end{array}
$$

Depending on whether the relation is considered 'derived' or 'given', it must be given the appropriate status before it is added to the database of relations. In the case that the relation is 'given', it must also be supplied with some explicit information. A possible implementation is that the user could be presented with all the information of the individual, non-derived *rel_step*'s and be given the opportunity to edit the information before the new relation is stored.

$$
\begin{array}{|l}
getstat : ((\mathbb{P}\ TransRel) \times Relation \times Exp \times Exp) \twoheadrightarrow Status \\
\hline
is\_rel\_conn\_derived(rc, r)(e, e') \Rightarrow \\
\quad getstat(rc, r, e, e') = derived \\
\\
is\_rel\_conn\_given(rc, r)(e, e') \Rightarrow \\
\quad (\exists\, info : Info \bullet \\
\qquad getstat(rc, r, e, e') = given(info))
\end{array}
$$

References

[1] R.J.R. Back. *On the Correctness of Refinement Steps in Program Development.* Ph.D. Thesis A-1978-4, Dept. of Computer Science, University of Helsinki, 1978.

[2] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[3] R.J.R. Back. Refinement diagrams. In J.M. Morris and R.C. Shaw, editors, *Fourth BCS Refinement Workshop.* Springer–Verlag, 1991.

[4] R.J.R. Back, J. Hekanaho, and K. Sere. *Centipede – A Program Refinement Environment.* Reports on Computer Science and Mathematics Series A, No. 139, Dept. of Computer Science, Åbo Akademi, 1992.

[5] J.S. Fitzgerald. Unification: Specification and development. In C.B. Jones and R.C. Shaw, editors, *Case Studies in Systematic Software Development.* Prentice–Hall, 1990.

[6] RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice–Hall, 1992.

[7] J. Grundy. Window inference in the HOL system. In P.J. Windley et al, editor, *International Tutorial and Workshop on the HOL Theorem Proving System and its Applications.* IEEE Computer Press, August 1992.

[8] C.C. Morgan. *Programming from Specifications.* Prentice–Hall, 1990.

[9] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Prog.*, 9(3):298–306, 1987.

[10] M. Naftalin. Informal strategies in design by refinement. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94.* Springer–Verlag, October 1994.

[11] D.S. Neilson and I.H. Sorensen. *The B-Technologies: A system for computer aided programming*. B-Core(UK) Ltd., 1994.

[12] P.J. Robinson and J. Staples. Formalising the hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, February 1993.

[13] J.M. Spivey. *The Z Notation - A Reference Manual.* Prentice–Hall, 1989.

[14] S. Vadera. Building a theory of unification. In C.B. Jones and R.C. Shaw, editors, *Case Studies in Systematic Software Development.* Prentice–Hall, 1990.

[15] J.L.A. van de Snepscheut. *Proxac: an editor for program transformation.* Jan 183-0, Caltech, 1993.

[16] T. Vickers. An overview of a refinement editor. In *5th Australian Software Engineering Conference*, May 1990.

[17] J. von Wright. Program refinement by theorem prover. In *Sixth BCS-FACS Refinement Workshop*, January 1994.