This document specifies *Memphis*, an extension of C/C++ [1, 2], that supports the definition and processing of symbolic data such as abstract syntax trees in compilers. 

*Memphis* provides a new kind of type declarations to describe such data via recursive definitions in a grammatical style. It also introduces a new statement to process such structures via rules that are selected by tree pattern matching. These concepts are known from functional languages [3] and modern compiler construction tools [4].
1 Introduction

Memphis covers C/C++ and adds domain definitions (extending the syntactic category declaration) and match statements (extending the syntactic category statement).

2 Domain Definitions

domain-definition:
   domain identifier { variant-declaration-list }
variant-declaration-list:
   variant-declaration
   variant-declaration variant-declaration-list
variant-declaration:
   identifier ( argument-specification-list_opt )
argument-specification-list:
   argument-specification
   argument-specification , argument-specification-list
argument-specification:
   identifier identifier

A domain definition of the form
   domain Dom { 
       Variant_1
       ...
       Variant_n
   }
introduces a data type Dom, called a domain type, that has n variants.

For example,

   domain Statement { 
       assignmentStatement (Expression target, Expression source)
       whileStatement (Expression condition, Statement body)
   }

declares a domain type Statement with two variants.

A variant declaration of the form
   Kind ( Type_1 Name_1 , ... , Type_k Name_k ) ;
introduces a data type Kind_subtype, called a variant type.

A value of this type is a pointer to a structure tagged as Kind. This structure has k fields, where Type_i is the type and Name_i is the name of the i-th field.

For example,
whileStatement (Expression condition, Statement body)

introduces a variant type whileStatement_subtype whose values are pointers to a structure that is tagged as whileStatement and has a field condition of type Expression and a field body of type Statement.

Domain declaration may be mutually recursive and may be given in any order (there are no forward declarations).

Values of a variant type are also values of the corresponding domain type. Where an expression of the corresponding domain type is valid, an expression of a variant type is also valid.

For example, with the declarations

    Statement st;
    assignmentStatement_subtype asg;

the assignment

    st = asg;

is valid. An assignment

    asg = st;

is not valid (st could hold a whileStatement_subtype value).

A value of a variant type introduced by

    Kind (Type1 Name1, ..., Typek Namek);

is constructed by an expression of the form

    Kind(E1, ..., Ek)

where \( E_1, ..., E_k \) are expressions of types \( Type_1, ..., Type_k \).

Its value is the pointer to a newly created structure that is tagged as \( Kind \) and has fields \( v_1, ..., v_k \), where \( v_1, ..., v_k \) are the values obtained by evaluating the expressions \( E_1, ..., E_k \).

For example, let \( a \) and \( b \) be variables of type Expression, then

    assignmentStatement(a, b)

constructs a value of the variant type assignmentStatement_subtype (that is also a value of the domain type Statement). Its fields are set to the values of \( a \) and \( b \).
If $x$ is an expression of a variant type, then a field $f$ of the structure to which $x$ points may be designated by $x \rightarrow f$. This designator may be used to access and modify the value of the field.

For example, if `asg` is a variable of type `assignmentStatement_subtype` then

```plaintext
expr = asg->source;
```

assigns the value of the `source` field to `expr` and

```plaintext
asg->source = expr
```

sets the `source` field to the value of `expr`. □

### 3 Match Statements

A match statement of the form

```plaintext
match $e_1, \ldots, e_n \{ \rule_1 \ldots \rule_k \}
```

is elaborated as follows: First, the expressions $e_1, \ldots, e_n$ are evaluated, yielding values $v_1, \ldots, v_n$. Then the rules $\rule_1, \ldots, \rule_k$ are elaborated in the given order until one succeeds. It is a checked run time error when all rules fail.
A rule has the form

\[
\text{rule } p_1, \ldots, p_n : \\
S_1 \\
\vdots \\
S_m
\]

It is elaborated as follows. First, the values \(v_1, \ldots, v_n\) are matched against the patterns \(p_1, \ldots, p_n\). If this fails, the rule fails. Otherwise the statements \(S_1, \ldots, S_m\) are elaborated and the rule succeeds.

If an expression \(e_i\) is of type \(T_i\), then a pattern \(P_i\) is said to appear on a position of type \(T_i\).

For example, consider a function \(\text{CheckLists}\) that checks two values \(\text{actual}\) and \(\text{formal}\) of a domain type \(\text{List}\).

\[
\text{CheckLists}(\text{List } \text{actual}, \text{List } \text{formal}) \\
\{ \\
\text{match } \text{actual}, \text{formal} \{ \\
\text{rule } \text{list}(\text{actHd}, \text{actTl}), \text{list}(\text{formHd}, \text{formTl}) : \\
\text{CheckMembers}(\text{actHd}, \text{formHd}); \\
\text{CheckLists}(\text{actTl}, \text{formTl}) \\
\text{rule } \text{empty}(), \text{empty}(): \\
; \\
\text{rule } \text{empty}(), \text{list}(\text{formHd}, \text{formTl}) : \\
\text{Error("actual list too short")}; \\
\text{rule } \text{list}(\text{actHd}, \text{actTl}), \text{empty}() : \\
\text{Error("formal list too short")};
\}
\}
\]

If both lists contain at least one member (\(\text{actHd}\) and \(\text{formHd}\)), the first rule is selected that checks these members and recursively processes the tails of the lists.

If both lists are empty, the second rule is selected and no further action is being performed.

If one is empty and the other is not, the third or the fourth rule emits a corresponding error message.

A value \(v\) is matched against a pattern \(p\) as follows.

If the pattern is an identifier \(id\), the the matching always succeeds.

\(id\) is implicitly declared as a variable local to the rule. It has the type of the corresponding position and is initialized with the value \(v\).

For example, if the value is \(\text{assignmentStatement}(s, t)\)
and the rule heading is

\[ \text{rule } st: \]

the matching succeeds. \( st \) is set to \text{assignmentStatement}(s,t). \( st \) is a variable of type \text{Statement} that is declared local to the rule.

If the pattern has the form

\[ f(p_1, \ldots, p_n) \]

then the matching succeeds if \( v \) is a value of the form

\[ f(v_1, \ldots, v_n) \]

and matching the values \( v_1, \ldots, v_n \) against the patterns \( p_1, \ldots, p_n \) succeeds; otherwise the matching fails.

If the pattern appears on a position with type \( T \) then \( T \) must be domain type with a variant of the form

\[ f(T_1 I_1, \ldots, T_n I_n) \]

The subpatterns \( p_1, \ldots, p_n \) appear on positions with \( T_1, \ldots, T_n \).

For example, if the value is

\[ \text{assignmentStatement}(s,t) \]

and the rule heading is

\[ \text{rule assignmentStatement(left, right)}: \]

then the matching succeeds and \( left \) and \( right \) are set to \( s \) and \( t \), respectively. \( left \) and \( right \) are variables of type \text{Expression} that are declared local to the rule.

Finally, if the pattern has the combined form

\[ f(p_1, \ldots, p_n)id \]

then the matching succeeds if \( v \) is a value of the form

\[ f(v_1, \ldots, v_n) \]

and matching the values \( v_1, \ldots, v_n \) against the patterns \( p_1, \ldots, p_n \) succeeds; otherwise the matching fails.

\( id \) is implicitly declared as a variable local to the rule. It has the variant type \text{f subtype} and is initialized with the value \( v \).

The type constraints of the second form also apply to this form.

For example, if the value is

\[ \text{assignmentStatement}(s,t) \]

and the rule heading is

\[ \text{rule assignmentStatement(left, right) st}: \]

then the matching succeeds and \( left \) and \( right \) are set to \( s \) and \( t \), respectively. \( left \) and \( right \) are variables of type \text{Expression} that are declared local to the rule.

\( st \) is set to \text{assignmentStatement}(s,t).
$st$ is a variable of type `assignmentStatement_subtype` that is declared local to the rule.

Because $st$ is declared as of a variant type its fields can be accessed as in $st->\text{target}$.

A pattern

$f (i_{d1}, \ldots, i_{dn})$

can be abbreviated by

$f()$

if $i_{d1}, \ldots, i_{dn}$ are not used.

For example, the rule heading

```plaintext
rule assignmentStatement (left, right) st :
```

can be written in the form

```plaintext
rule assignmentStatement () st :
```

and $st -> \text{target}$ could be used instead of $\text{left}$.

The rules

```plaintext
rule empty(), list(formHd, formTl) :
   Error("actual list too short");
rule list(actHd, actTl), empty() :
   Error("formal list too short");
```

can be written as

```plaintext
rule empty(), list() :
   Error("actual list too short");
rule list(), empty() :
   Error("formal list too short");
```

since the field names are not used in the rule bodies.

4 File Clauses

```plaintext
file-clause:
   with identifier ;
```

If a source file refers to domain declarations of another source file, the referring file must contain a `file-clause` of the form

```plaintext
with name ;
```

where `name` is the base name of the referred file.

This is equivalent to textually including the domain declarations of the referred file at the place of the `file-clause`. 
For example, assume that a file “ast.m” contains domain declarations that specify abstract syntax trees. A file “walker.m” that contains match statements to process these trees must provide a clause

\[
\text{with ast ;}
\]

References

[1] Brian W. Kernigham, Dennis M. Ritchie:

[2] Ellis Stroustrup:

[3] Robert Harper, Robin Milner, Mads Tofte:

[4] Friedrich Wilhelm Schröer:
Gentle, In Studien der GMD 166, German National Research Center for Information Technology (1989)

See Also

Memphis C/C++, A Language for Compiler Writers
Memphis User Manual
memphis.compilertools.net