An Approach to Designing Menu-based Languages

Alan M. Durham
Universidade de São Paulo
durham@ime.usp.br
Ralph E. Johnson
University of Illinois
johnson@cs.uiuc.edu

Abstract

One of the few parts of a compiler that has still not been automated is defining the data types used by the run-time system. We developed a tool that lets a compiler writer specify formats o data types by filling in forms and be selecting options from menus. The tool then generates those parts of the compiler related to data type formats. The approach we took to designing the system is general and can be applied to other kinds of probems. This approach results in an object-oriented application generator, and is a natural result of designing an object-oriented framework.

In this paper we describe the problem, how we developed the solution, and the lessons learned in the process.

1 Introduction

Visual programming languages are one way of making software development easier [Shu 88]. Many problem domains have graphical notations that are widely used, and a visual programming language can be based on that notation [Casner 90]. But many problem domains do *not* have widely used graphical notations. What can you do if you need to make a graphical specification language for some specialized problem domain, and it does not have a standard graphical notation?

Another way to automate the process of software development is to build frameworks for each problem domain [Johnson 88a]. Some of these frameworks can be used to build program generators with graphical front ends. To enable the creation of program generators, the framework has to be "black-box" enough, that is, it has to be based on components that hide all the details of their implementa-

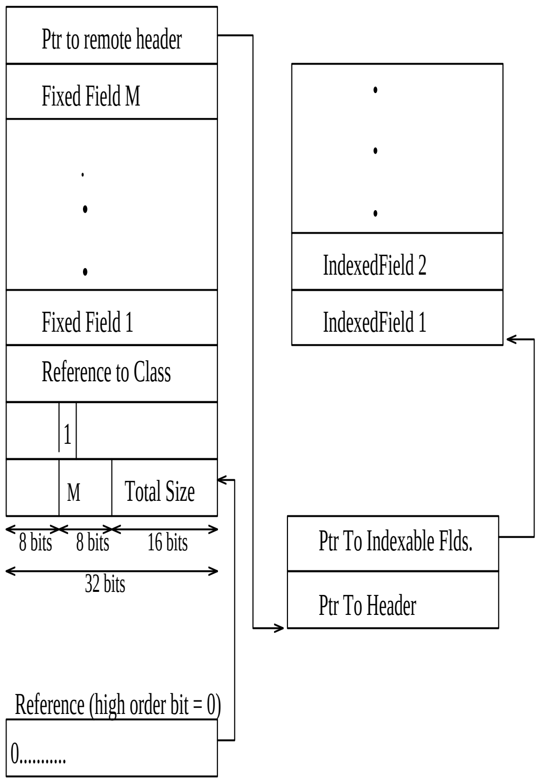

Figure 1: Run-time format of Big Objects in Tektronix ST-80

tion, presenting an abstract model to the final user. A common strategy to achieve such black box implementations is to first figure out how to represent the components of a solution to the problem domain. This phase usually requires iteration and refactoring [Opdyke 90].

The first phase should produce a set of well defined components that provide an abstract model for a solution in the problem domain. The second phase consists of building a graphical front end for the components developed in the first phase.

This paper presents an example of the approach described above, applied to the problem domain of run-time system implementation. We describe the problem, the process of developing the "black-box" components, and the lessons we learned, including some guidelines for building components that can be attached to a graphical interface.

2 The Problem

Run-time representation of data types can be arbitrarily complex. It can range from machine-operable data, like integers in imperative languages, to complex data structures, like objects in object-oriented languages. Data type representations can be composed of many data blocks, or can have tag fields or tags that are part of other fields, like pointer fields. A good example of this complexity is the Tektronix representation of large objects in Smalltalk-80 (figure 1).

Complex data types are generally implemented by a set of procedures, called primitive operations. However, primitive operations make a complex specification for run-time systems. We wanted to find an easier model for specifying run-time systems, in particular for specifying data types. This paper describes how we developed a system for specifying run-time data type format that uses only menus and templates.

Our work was developed as part of the TS compiler [Johnson 88b], a compiler for typed Smalltalk. We were concerned specifically with run-time system implementation, which includes the implementation of data types. Separating format description from the description of the semantics of primitive operations seemed to lead to more modular code. Format description seemed to us a domain enclosed enough to build a visual tool.

This paper describes how we developed such a tool and the design principles we learned in the process. Our final goal was to produce an easy-to-use tool to solve the problem. A tool based on menus and templates seemed ideal. Using an object-oriented approach encouraged us to classify the various elements of the problem. This classification and the associate class hierarchy enabled us to create the desired specification tool for data type format.

3 Obtaining a solution

The original design had a set of routines to generate code for:

- accessing data for the primitive operations,
- storing data,
- doing run-time type checks using the internal tags,
- moving objects in memory,
- allocating new objects,
- tracing pointers for garbage collection.

The code generation routines had to be written by the compiler writer. The routines encoded details about the format of data types. The system did not impose modularity, modular code would be a consequence only of the compiler writer's programming skills. A cure requires a model for formats that would show us the primitive elements of a format specification.

The first step was to find what the compiler needed to know about data type formats. The code generation methods depended on:

- the position of each atomic data item, which we will call a *field*;
- how fields are grouped together;
- the format of a data type reference, that is, the machine word that constituted the initial "handle"to the run-time representation of the data type;
- the location of pointers to instances of other data types;
- the location and values of tags;
- the location of indexed data;
- the location of internal pointers, that is, pointers that point to other parts of the same instance of the data type;
- the format of pointers, which are either absolute addresses, relative addresses, or indexes in a table.

It became clear that describing the format of a data type could be done by describing the format of the individual fields, and describing how these fields were grouped in data blocks and in the reference. So, we divided a description of the run-time format of the data types into three parts:

- describing the *fields*,
- describing the *data blocks*,
- describing the *reference*.

A *field* is the atomic information unit inside a primitive data type. Instances of a primitive data type can hold more than one unit of information, and therefore can have many fields. Fields are grouped in one or more contiguous blocks of memory, called *data blocks*. A *reference* is the initial machine word used to designate the instance of that data type. This reference can contain the whole data object, or can contain a pointer to the rest of the representation. The reference itself can be divided into fields, including tag fields.

Accessing and updating a field requires knowing the position of the field. Since each field belongs to a data block or a reference, only its relative position and size was needed. Also, fields can hold different kinds of information:

- references to other instances of data types,
- internal data used by the primitive operations,
- pointers to other parts of the data type's representation.

We generate access and update code for all fields. Fields that contain references may need different update code due to memory management. Fields that contain references also need to be considered when generating routines that perform reference tracing. Pointers to data blocks are also important for memory management routines, but are treated differently than references. Also, pointer fields may contain an absolute address, a displacement, or an index in a pointer table. For all types of pointers, we need to have a description of the associated data block, that is, a description of the contents of the data block it points to. Some information, however, is specific to the pointer type:

- a relative pointer field depends on the base address,
- an index to a pointer table depends on the table base and on the size of the entries.

It is easy to see that different specifications need to be maintained, depending on which type of data the field holds at run-time. We tried to group these specifications in sets, which led to the following hierarchy of field types:

- **field**
  1. **data field**
  2. **pointer field**
     (a) **relative pointer**
     (b) **table pointer**
  3. **reference field**

Both references and data blocks hold one or more fields. The difference is that the reference is always the starting point of any access path to a field. Data blocks, on the other hand, are always reached through a pointer field that is either part of another data block or part of the reference. This field needs to be recorded for each data block, if we want to generate access code to it. That means accessing fields inside data blocks is somewhat different from accessing fields inside references. This lead us to create a classification for field holders:

- **field holder**
    1. **data block**
    2. **reference**

Based on the classification above and on the information we needed to know about each component of a data object, we created a set of classes to describe data type formats. These classes reflect the type hierarchies we have specified in this section. The instance variables of each class store the information necessary for the associate type. The compler uses the methods of each class to generate code.

4 A class hierarchy of format descriptors

A primitive data type's format description specifies its fields, and how the fields are bundled together in the data blocks and in the reference. There are three basic classes of descriptors:

- A *PDTDescriptor* defines the format of a data type, and also define the basic code generation protocol.
- A *PDTFieldHolderDescriptor* describes a field holder that can be either a reference or a data block. It stores the list of fields that are managed by the field holder. This abstract class has two concrete subclasses, *PDTDataBlock* and *PDTReference*. Each PDTDataBlock keeps track of the descriptor of the field that it is in.
- A *PDTField* describes the fields of a data type.

PDTField has many subclasses, one for each field type. The complete PDTField class hierarchy is (in parenthesis we list the instance variables):

- PDTSingleField (displacementInDataBlock firstBit lastBit mask tagValue)
    - PDTReferenceField ()
    - PDTPointerField (dataBlockDescriptor)
        - PDTIndirectPointerField (tableBase entrySize isIndex)
        - PDTRelativePointerField (baseField)
    - PDTDataField
- PDTRepetitionField (repeatedField)
    - PDTFixedRepetitionField (repetitionNumber)

– PDTVariantRepetitionField
(sizeFieldName)

Repetition fields handle indexed access inside a data type. Each of these fields needs the total number of repetitions and the appropriate specification information, depending on the type of information stored at runtime. Since the instances of a repeated field hold the same type of information that single fields do, the instances of PDTRepetitionField have a component that describes a single field. This avoids creating a subclass for each type of repeated field or using multiple inheritance. PDTRepetitionField only provides the indexing capabilities, everything else is the responsibility of its components.

The instance variables "mask" and "tagValue" of PDTSingleField were introduced to handle tags. Initially, we considered creating a "tag field" type. However, sometimes tags are an integral part of the information contained in some other field[1]. The variable "mask" gives the location of the tag bits, and "tagValue" gives their value.

The subclasses of PDTField, along with the DataBlock descriptors and the PDTReference descriptors, generate the code for data type access and update operations. The PDTDescriptor holds all the format information for a data type. It can generate code for reading or writing any field in the data type.

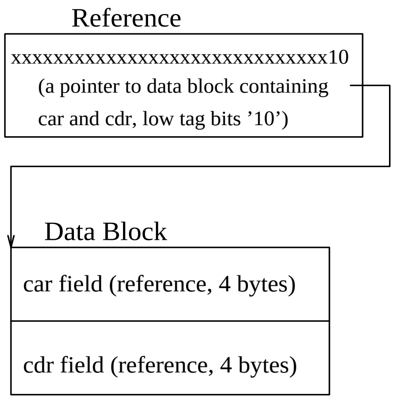

Figure 2: Cons cell design

Figure 2 shows a typical representation of a LISP *cons* cell, with the reference containing a pointer to a data block that contains the *car* and the *cdr* fields. Both *car* and *cdr* are 4 bytes long. The reference has two tag bits, with contents '10'. Figure 3 shows how this data type is specified with our descriptors.

All code generation requests are directed to an instance of PDTDescriptor. The compiler has a set of PDTDescriptors for a given run-time. It generates code by selecting the appropriate instance of PDTDescriptor and asking it to generate code to read or write its fields. The PDTDescriptor finds the descriptor of the field to be accessed/updated and asks it to generate code for the appropriate operation. The field descriptor will interact with the data block descriptors, the reference descriptor, and, eventually, with other field descriptors, to generate the appropriate code sequence. This interaction is neces-

[1] An example of this can be seen in the Tektronix Big Object from figure 1: the pointer field has a high bit tag with value 0. There is no need for field extraction methods to get the pointer value, the tag is part of the pointer. Low-bit tag with value 0 is also a common tag.

sary because the field's location information is distributed among the descriptors. Each field descriptor knows its relative position inside its data block. Each data block knows the field that it is in. The reference descriptor stops this delegation process.

The solution to data type format specification presented here has some advantages:

- the specifications are now more modular and positioning information is explicit;
- format specification can be easily reused by copying the objects from previous specifications;
- writing methods that generate format related code was made easier, as all format related information is now explicitly stored and classified; the new methods are always in the description classes.

5 A visual language to specify data types

Any run-time system data format can be described using only the classes in the previous section. It is not necessary to make more subclasses. This makes it easy to build a language to describe data type implementations using only menus and templates. Menus are used to select the classes to be instantiated. Templates are used to parameterize the instances and to establish links between them, such as the

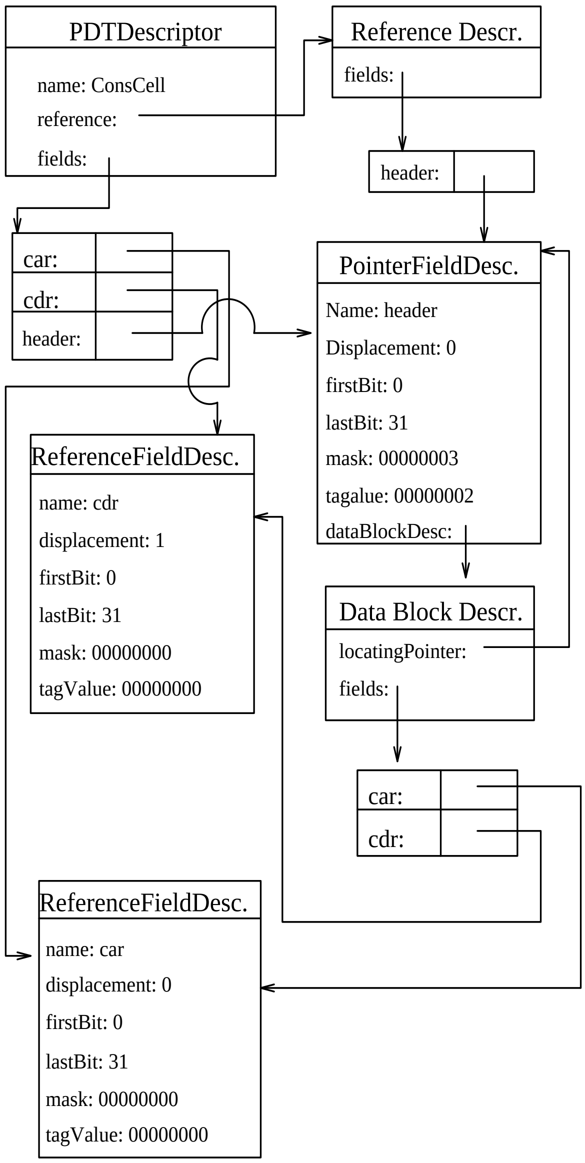

Figure 3: Specification of cons cell design

links between a data block and its fields. To permit the specification of links, Each descriptor has a name and so a link can be specified either by typing the name of the descriptor (as in the case of a relative pointer) or by selecting a descriptor from a list (as when selecting the block to which a field belongs).

We are currently using a version of the system described here as part of a run-time system specification tool. Instances of these classes are created and managed by a tool called PrimitiveDataTypesBrowser, which coordinates specification of all aspects of data type implementation, including the semantics of the primitive operations. Users of PrimitiveDataTypesBrowser can specify the format of data types by using only menus and templates, due to the fact that all code is generated automatically from the information stored in the instance variables of the format description classes. Figures 4 and 5 show examples of such templates.

6 An approach to develop a menu and template based specification tool

The first step in developing a domain specific visual language is to clearly delimit the problem domain. In our case the problem domain was the specification of data type formats. Part of this step is collecting examples of the kinds of things you want to express in the language. In our case, we were working with the set of data types of languages like Smalltalk, Lisp,

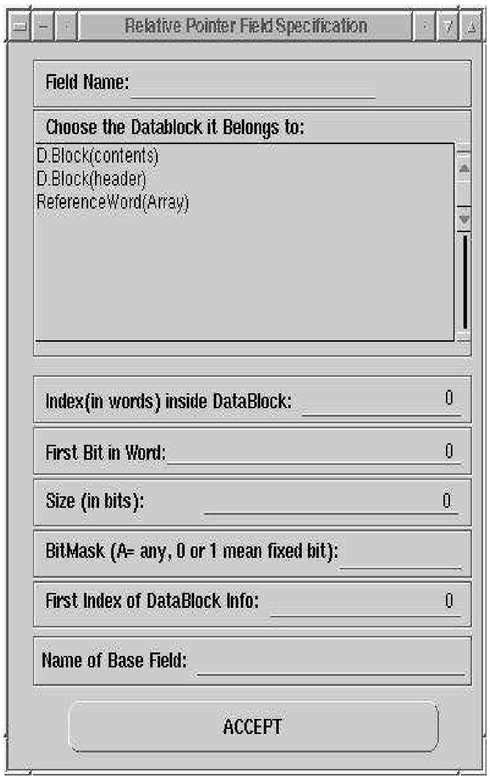

Figure 4: Template for reference field

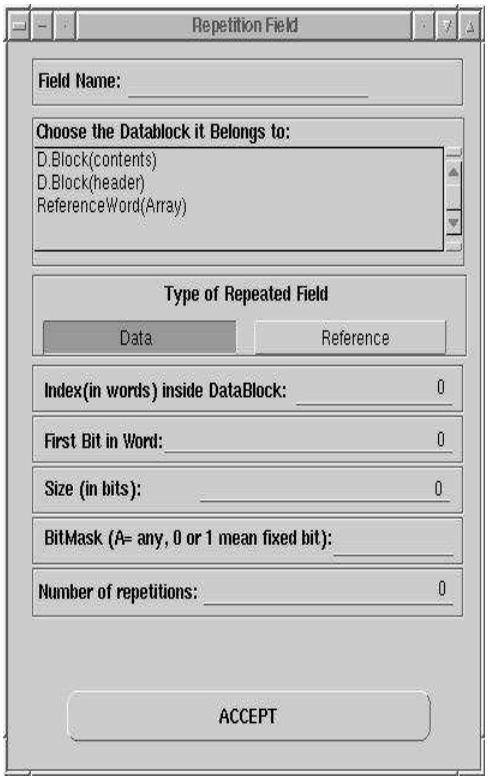

Figure 5: Template for multiple field

and Scheme. Looking at examples is a good way to understand the breadth of a problem domain, and it also is useful for the next step.

The next step is to develop an object-oriented representation of the problems to be solved. In our case, it was the specification of the data type formats. We did not worry about the code generation algorithms, but instead focused on the kind of objects needed to specify the format. We tested our representation by trying it out on several data types.

The next step is to make this "language" executable. We organized the objects into class hierarchies and tried to implement the code generation algorithms. The code generation algorithms would traverse the object structures. The code generation algorithms followed the "Interpreter" pattern [Gamma 95]. It is important to use the examples gathered early as test cases.

Generating code from the information stored in the object structure was not trival, and required changing the implementation of the classes. For example, we initially did not have a class for the PDTReferenceField because it is common for references to be only a pointer, so we just put its functionality in the PDTPointerField class. However, we ended up with different sets of methods for each case, even though all the code was in one class. Separating the cases made the design simpler. This is the kind of information that is hard to know in advance.

Finally, once all the classes have been written, build the visual tools to specify solutions to the problem domain. Use menus to choose the classes to be instantiated, and use templates to parameterize the instances. Since all instance variables are either simple values like numbers or strings or references to other instances of the specification classes, these templates are easy to build.

7 Conclusion

This paper presents an approach to develop visual tools. We showed how we were able to create a tool to specify data type format by using an object oriented approach. This approach is generic and can be used to develop similar tools for other problem domains.

This approach lets you develop a visual language as you develop its implementation. It is not the right approach to use when you have been giving a visual language and are just supposed to implement it. But if you do not have a visual langauge for your problem domain then it can be a good approach.

8 Bibliography

[Casner 90] Stephen Casner Building Customized Diagramming Languages. *Visual Languages and Visual Programming*, pages 71-95. ed. Shi-Kuo Chang, Plenum Press, 1990.

[Johnson 88a] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22-35, June/July 1988.

[Johnson 88b] Ralph E. Johnson, Justin O. Graver and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *Proceedings of OOPSLA '88. Object Oriented Programming Systems, Languages, and Applications.* pages 18–26. November 1988. Printed as SIGPLAN Notices, 23(11).

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[Opdyke 90] William F. Opdyke and Ralph E. Johnson. Refactoring: an aid in designing application frameworks and evolving object-oriented system. In *SOOPPA Conference Proceedings*, pages 145–161, Marist College, Poughkeepsie, NY, September 1990. ACM Press.

[Shu 88] Nan C. Shu. *Visual Programming.* Van Nostrand Reinhold Co, NY, 1988.