SOFTWARE TEMPLATES

Dennis M. Volpano
B.S., University of Wisconsin Green Bay, 1980
M.S., Purdue University, 1982

A dissertation submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science & Engineering

October, 1986

The dissertation "Software Templates" by Dennis M. Volpano has been examined and approved by the following Examination Committee:

Richard B. Kieburtz
Professor
Thesis Advisor

Richard G. Hamlet
Professor
Examination Committee Chairman

James L. Hein
Associate Professor
Portland State University

Mayer D. Schwartz
Computer Research Laboratory
Tektronix, Inc.

ACKNOWLEDGMENTS

I wish to thank Dr. Richard B. Kieburtz for his guidance, inspiration, and patience during the course of this research. Thanks also to Mark Foster for his assistance and to Mark Ballard for many stimulating discussions.

I would also like to acknowledge the members of my dissertation committee for their careful review of this dissertation and for their insightful comments and suggestions. Lastly, I am deeply indebted to my wife Wendy for her support throughout my years of graduate study.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Software Templates

Dennis M. Volpano, Ph.D.
Oregon Graduate Center, 1986

Supervising Professor: Richard B. Kieburtz

Software reuse is widely recognized as a key to improving both programmer productivity and the quality of software produced. However, software components found in existing libraries often cannot be reused because the algorithms they realize have been encoded in terms of particular implementations. An approach to reusability is presented where algorithms and implementations are specified separately. An algorithm is specified as a typed polymorphic function called a *software template*. Templates are defined over values of abstract data types whose implementations are specified separately and catalogued. When a template's data types are bound to catalogued implementations, the template is *instantiated* to a program component tailored to the chosen implementations. Different implementations of an algorithm can be achieved by merely rebinding the data types of its template specification to different catalogued implementations.

There are four primary steps to instantiating a template. First, the template is transformed into tail-recursive form through the introduction of continuations. Implementation bindings for the template's data types are then propogated

throughout the tail-recursive template by a type inference. Next, the tail-recursive template is translated into an intermediate, imperative-language code from which concrete code is generated in the final step. The result is a program component fully tailored to those implementations chosen to implement the abstract data types. Correctness of an instantiated template is established by separately verifying the correctness of the template and the implementations of its data types.

1. INTRODUCTION

The programming industry has been one of the fastest growing professional occupations, employing about 3.25 million programmers worldwide in 1983 [30]. In 1983 alone, their efforts amounted to over 6 billion lines of source code. If the current trend in the growth of the programming profession continues then by 1990, we will see an annual output of around 15 billion lines of code. With an output as prodigious as this, one naturally asks if there are that many applications calling for so much new code. Studies [30, 35] reveal that for many applications, programs usually have operations in common that could be standardized, strongly suggesting that much of this new code could be avoided through software reuse.

The value of software reuse has spurred a great deal of research [7, 10, 25, 36, 43] including a workshop devoted solely to reusability. To quote Alan Perlis, the general chairman of this workshop: "reusability has taken on an increased sense of urgency in the light of the growing demand for software and escalating software costs" [48]. Software reuse is widely recognized as a key to improving both productivity in software development and the quality of software produced. Productivity gains come from spending less time in developing new software, while quality is enhanced through utilizing software already proven reliable.

If programming is perceived as an activity in which a few basic algorithms are repeatedly implemented with perhaps slightly different implementation constraints [15, 53], then why do studies [30, 35] continue to reveal little software reuse? There

are many reasons, but one primary reason is related to specification. Software components are often expressed in conventional, imperative languages, making it very difficult to specify algorithms generically. Commitments to specific types and representations of data are made that often hamper efforts to reuse the components. It is difficult for example, to specify a sorting component in most conventional, imperative languages without committing to a type or representation for the objects being sorted. Obviously, commitments appropriate for one application may not be suitable for another.

Although software components expressed in an imperative language defy reuse, they do have an evident advantage in efficiency over more general specifications. Efficiency comes as a result of making commitments, a step that is in direct conflict with reusability. For instance, committing to an array sequence representation affords accessibility to sequence elements in constant time. Components expressed in an imperative language can be optimized to exploit properties of the imperative model of computation such as incremental updates and side effects. So naturally we ask are there general specifications of algorithms that can be reused without compromising efficiency, and if so, what do they look like?

1.1. Software Templates

In this dissertation, a programming methodology is presented in which reusability and efficiency can coexist. It is based on a factorization of the programming process into two separate activities, algorithm and implementation specification. Algorithms are specified in a general way as typed polymorphic functions called *software templates.* Templates are defined over values of abstract data types whose

implementations are specified separately. These separate programming activities culminate in a library of templates for standard algorithms, and a library of implementations for commonly-used abstract data types. The template user binds the data types over which a template is defined to catalogued implementations. The template is then automatically translated into an efficient program component tailored to the chosen implementations, a process called *template instantiation.* Different implementations of an algorithm can be achieved by merely rebinding the data types of its template specification to different catalogued implementations.

A template is a typed function defined in terms of the operators of one or more abstract data types. For example, suppose we wish to specify a template *rev* to reverse the elements of a sequence. An abstract sequence type is defined recursively as a signature algebra in Figure 1.1 [19]. The operators *nil*, *apndl* and *apndr* are constructors used to construct values of type *seq*. That is, a value of type *seq* is the empty sequence *nil*, or a sequence constructed from the left with *apndl*, or a sequence constructed from the right with *apndr* [5, 61]. The type *seq* is polymorphic in that the

$$
\begin{aligned}
&seq\,(*) = nil \;+\; apndl\,(*,\; seq\,(*)) \;+\; apndr\,(seq\,(*),\; *)\\
&\qquad hd\,(apndl\,(h, \_)) = h;\\
&\qquad tl\,(apndl\,(\_,\; t)) = t;\\
&\qquad hdr\,(apndr\,(\_,\; hr)) = hr;\\
&\qquad tlr\,(apndr\,(tr, \_)) = tr;\\
&\qquad null\,(nil) = true;\\
&\qquad null\,(apndl\,(\_, \_)) = false;\\
&\qquad null\,(apndr\,(\_, \_)) = false;\\
&\qquad apndr\,(apndl\,(x,\; y),\; z) = apndl\,(x,\; apndr\,(y,\; z))
\end{aligned}
$$

Figure 1.1. Polymorphic sequence data type

type of sequence objects is left unspecified as conveyed by the type variable '$*$'. The template *rev* might be defined recursively by

$$rev = \lambda x.\ null\,(x) \rightarrow nil;\ apndr\,(rev\,(tl\ x),\ hd\ x)$$

It has the mapping type $seq\,(*) \rightarrow seq\,(*)$. The operators *null*, *hd* and *tl*, axiomatized in Figure 1.1, are discrimination and elimination functions over sequences.

Implementations of abstract data types are specified in an imperative language by an implementor, someone conversant with the imperative language. Any imperative language with a type definition facility can be used. An abstract data type is implemented by first selecting a data representation for values of the type, and then implementing its operators, not necessarily all of them, in terms of the selected data representation. For example, two very common implementations of the sequence data type are array and file implementations. In the array implementation, a sequence is given a bounded representation as a fixed-size array. Consequently, only those finite sequences whose lengths do not exceed the array size can be represented. In the file implementation, a sequence is given an unbounded representation as a file, enabling finite sequences of arbitrary length to be represented.

When a template's data types are bound to particular catalogued implementations, it is instantiated to a program component expressed in the imperative language of the chosen implementations. Binding is accomplished by an instantiation context in which a template user makes certain commitments. There are actually two kinds of commitments that can be made. The first kind, one we have already alluded to, is an *implementation* commitment. A template user selects certain implementations of those abstract data types over which a template is defined. Some of these data types

may be polymorphic, calling for another kind of commitment, a *parameter-type* commitment. Here a template user binds the parameters of polymorphic data types to specific types. Both kinds of commitments are conveyed by a *representation type* expression. If T is a polymorphic data type of one parameter then the representation type expression *repty* $(T(\alpha), I)$ indicates the selection of an implementation I of the type T, and the binding of T's parameter type to α.

A request to instantiate a template is made by specifying an instantiation directive in an application program. A directive contains the name of the template to be instantiated and one or more *abstract* variables whose declarations establish a context for the instantiation. Abstract variables will serve as input and output variables of the program component generated by the instantiator [60]. For example,

assign$(s', rev\ s)$

is a directive that instructs the instantiator to tailor the template *rev* to a program component that binds the reverse of sequence s to s'. The kind of tailoring that *rev* undergoes is determined by an instantiation context established by declaring the abstract variables s and s'. For instance, if *Array* is the name of a Pascal array implementation of the abstract data type *seq*, parameterized in array size, then the context established by the declaration

$$\textbf{repty}(seq(int), Array(64))\ \ s, s'; \tag{1.1}$$

forces *rev* in the directive, to be instantiated to a Pascal component that reverses an array s of up to 64 integers (*int*), and places the result in an array s'. By merely changing the context, *rev* can be instantiated to a different component, perhaps expressed in a different language. For instance, *rev* can be instantiated to a

component expressed in C that reverses a file s of characters and places the result in a file s' with the declaration

$$\textbf{repty}(\mathit{seq}(\mathit{chr}), \mathit{File})\ \ s, s';$$

where *File* is a C file implementation of *seq*. The template *rev* can be reused in many different instantiation contexts. As we shall see, only the typing of a template specification varies across different contexts.

1.2. Related Work

An area of reusability research resembling template instantiation is *specialization* [54]. It is characterized by two steps: 1) establishing a context for a specification, and 2) translating the specification into a more efficient imperative form that exploits the context. The research of [13, 20, 54], three independent efforts in specialization, is outlined in this section.

The work of Gries and Prins [20] concentrates on the context-establishment step of specialization. A context is established for a specification through *implement directives*. They serve the same purpose as representation type expressions but are not nearly as expressive. For example, the context established by declaration (1.1) is established in their system through the following implement directives.

var s, s': *int seq*
implement s **by** *Array* (64)
implement s' **by** *Array*(64)

Although Gries and Prins acknowledge the importance of type polymorphism, their notation for establishing contexts does not support it. It is not clear for instance, how one can specify a floating point implementation say *Float*, of the parameter type *int*.

Using a representation type expression, this is accomplished by the declaration

$$\textbf{repty}(\mathit{seq}(\textbf{repty}(\mathit{int}, \mathit{Float})), \mathit{Array}(64)) \quad s, s';$$

Specifications are limited to the operations of abstract data types, so the translation step is mere replacement. That is, translating a specification merely involves replacing the specified operation by its implementation as prescribed by some context. For instance, if s denotes a sequence implemented by *Array* in some context, then the application $hd(s)$ is replaced with the implementation of *hd* in *Array*.

The programming language Ada with its generic facility, provides most of the capabilities developed by Gries and Prins. Their implementations (modules) can be constructed using generic packages which may be parameterized to import various types and functions [59]. However, their strategy is more flexible than generic package instantiation since modules need not be expressed in Ada.

Scherlis [54] has proposed an elaborate approach to specialization intended to exploit other properties of a context besides just implementation commitments. Intimate knowledge of a context is to be gleaned and used to guide the translation step, producing a very efficient imperative form. It is an intriguing strategy but its employment in developing software seems distant.

Darlington [13] has developed a semi-automatic means for specialization based on the transformation system described in [8]. A specification is a function defined over values of an abstract data type. A context is established for a specification by providing a representation function which maps values of an implementing data type to abstract values [27]. Translation into a more efficient form proceeds with user assistance but there is no final translation into an imperative language.

1.3. Dissertation Overview

The dissertation is organized in two parts. The first part (Chapters 2 and 3) addresses specification, and the second part (Chapters 4-7) addresses template instantiation. Chapter 2 describes the template specification of algorithms, and Chapter 3 describes the specification of abstract data type implementations in this methodology. Template instantiation, the process that bridges the separate specification of algorithms and implementations, comprises four major steps.

(1) Tail-recursive transformation. A template specification in general comprises a family of typed recursive function definitions. The first step of instantiation removes recursion, transforming a template into tail-recursive (iterative) form. The transformation is described in Chapter 4 where it is also shown to be complete.

(2) Representation type inference. The second step of instantiation propogates commitments made in an instantiation context, throughout the tail-recursive template produced in the first step. Commitments that affect implementations of operators are identified through a type inference at the level of representation type expressions. The type inference, described in Chapter 5, produces a representation typing of the tail-recursive template.

(3) Abstract imperative-language representation. The third step of instantiation translates the tail-recursive template into an intermediate, imperative language called *L* (*L*ocations). With the representation typing produced in the second step, implementations of operators are selected from an implementation environment. Selected implementations guide the translation, suggesting the use of either statements or expressions of *L* in implementing the operators. Chapter 6 is devoted to the

translation into L code. A major step of the translation is also proved correct.

(4) Concrete imperative-language representation. The final step of instantiation translates the L code generated in the third step into concrete code. The implementations of operators selected in the third step are used to expand statements and expressions of L to concrete form. Chapter 7 describes the generation of concrete code from L code.

Chapter 8 describes the current version of the software templates system. The dissertation concludes with Chapter 9 where the effect of the templates methodology on certain aspects of programming such as information hiding and debugging is examined. A notion of correctness in this paradigm is also discussed and future work is outlined.

2. TEMPLATE SPECIFICATION OF ALGORITHMS

The algorithm specification language is a typed first-order applicative language with polymorphic abstract data types. In this language, algorithms are specified as typed first-order functions called *templates*. Referential transparency (expression evaluation without side effects) is a desirable property of specifications since it makes them more amenable to formal reasoning and transformation. Polymorphic data abstraction is the key to making algorithm specifications reusable because two kinds of commitments that ordinarily hinder reusability can be postponed. That is, an algorithm can be specified without fully committing to either types or representations of the values over which it is defined.

2.1. Abstract Data Types

In the templates methodology, an abstract data type is an S-sorted Σ-algebra where S is a set of sorts each having a carrier, and Σ is a collection of operators among carriers, called the signature [18]. The axioms of the algebra serve to specify the semantics of the type's operators [21-23, 37, 38]. For example, in Figure 2.1 an abstract character sequence type is defined called *chrseq*. The signature is composed of the operators *nil*, *apndl*, *hd*, *tl*, and *null*. The definition specifies a union expression

$$nil + apndl(chr, chrseq)$$

composed of two constructor expressions [19]. Since the type name *chrseq* appears as an argument of *apndl*, the definition is recursive. The constructor expressions

prescribe how to construct values of type *chrseq*. These values are defined by terms freely generated from constants of the abstract type *chr*, and the operators *nil* and *apndl*. For instance, the following terms are values of type *chrseq*.

nil
apndl ('a', *nil*)
apndl ('b', *apndl* ('a', *nil*))

The operators *nil* and *apndl* are called constructors of the signature. A constructor is special in that its application is irreducible. The definition of *chrseq* also axiomatizes the remaining operators of the signature. The operator *null* tests for the empty character sequence *nil*, while the operators *hd* and *tl* extract the head and tail components of an *apndl*-constructed sequence.

2.1.1. Polymorphism

The type *chrseq* is an example of a monomorphic type. Some applications may require sequences to contain objects other than characters. Certainly a new type can be defined for each sequence of new objects, but a much better approach is to define a sequence type scheme in which the type of objects contained in a sequence is left unspecified. This way different instances of the same type scheme can be created for

---

$$
\begin{aligned}
chrseq &= nil + apndl(chr, chrseq) \\
&\quad hd(apndl(h, \_)) = h; \\
&\quad tl(apndl(\_, t)) = t; \\
&\quad null(nil) = true; \\
&\quad null(apndl(\_, \_)) = false;
\end{aligned}
$$

Figure 2.1. Character sequence data type

---

those applications that demand sequence objects of different types. A type scheme is a *polymorphic* type, one whose operators may be applied to operands of many distinct types. For example, the abstract data type *seq* defined in Figure 2.2 is a polymorphic type. Unlike *chrseq*, the type of sequence objects is left unspecified as conveyed by the type variable '*'. The union expression reveals three constructors, namely *nil*, *apndl* and *apndr*. Sequences can be constructed either from the left with *apndl*, or from the right with *apndr*. The signature also includes two elimination operators *hdr* and *tlr* to extract the right head and right tail components of an *apndr*-constructed sequence.

2.1.2. Predefined Types

The algorithm specification language provides four predefined primitive types *int*, *bool*, *chr* and *tok*, and two predefined compound types *union* and *pair*. Any integer constant has type *int*, while the boolean constants *true* and *false* have type *bool*. A character enclosed in single quotes has type *chr*, while a string (token) of one or more characters delimited by double quotes has type *tok*.

$$
\begin{aligned}
&seq\,(\star) = nil \;+\; apndl\,(\star,\; seq\,(\star)) \;+\; apndr\,(seq\,(\star),\; \star)\\
&\qquad hd\,(apndl\,(h,\; \_)) = h;\\
&\qquad tl\,(apndl\,(\_,\; t)) = t;\\
&\qquad hdr\,(apndr\,(\_,\; hr)) = hr;\\
&\qquad tlr\,(apndr\,(tr,\; \_)) = tr;\\
&\qquad null\,(nil) = true;\\
&\qquad null\,(apndl\,(\_,\; \_)) = false;\\
&\qquad null\,(apndr\,(\_,\; \_)) = false;\\
&\qquad apndr\,(apndl\,(x,\; y),\; z) = apndl\,(x,\; apndr\,(y,\; z))
\end{aligned}
$$

Figure 2.2. Polymorphic sequence data type

Let e be an expression of type α and e' be an expression of type β. A value of type $pair(\alpha, \beta)$ is formed by evaluating the expression $pr(e, e')$ where pr is a constructor. The components of a pair are accessed by the elimination operators *fst* and *snd* which satisfy the equations

$$fst(pr(x, y)) = x \qquad\qquad snd(pr(x, y)) = y$$

A value of type $union(\alpha, \beta)$ is formed by evaluating either the expression $inl(e)$, or $inr(e')$ where *inl* (inject left) and *inr* (inject right) are constructors. Injecting a value into a union type merely tags it indicating to which component type it belongs. The components of a union are accessed by the elimination operators *outl* and *outr* which satisfy the equations

$$outl(inl(x)) = x \qquad\qquad outr(inr(y)) = y$$

Both *outl* and *outr* are partial operators since the applications $outl(inr(y))$ and $outr(inl(x))$ are undefined. A discrimination operator *isl* is also provided to test if a union has been constructed by a left injection. It satisfies the equations

$$isl(inl(x)) = true \qquad\qquad isl(inr(x)) = false$$

2.1.3. Importation

Abstract data types must be imported for use in both algorithm and implementation specification. Importing a data type entails declaring a type for each of its operators. But unlike import declarations found in applicative languages [4, 9], abstract data types in the templates paradigm are imported in terms of *representation type* expressions. A data type by virtue of being abstract, can be implemented in many different ways, giving rise to a form of overload. Implementations however, can

be distinguished by their representations of abstract values. It is this information, how abstract values are represented, that is captured in a representation type expression. Consequently, these type expressions can be used to resolve the overloading of an abstract data type with multiple implementations.

A representation type expression can be defined inductively as follows. If T is a polymorphic type then the following expressions are representation type expressions.

(1) A representation type variable $\star n$ where n is a numeral.

(2) A mapping expression $e \rightarrow e'$ where e and e' are representation type expressions.

(3) *repty* $(T(e), R)$, where e is a nonmapping representation type expression and R is the name of a data representation for values of type T.

(4) *repty* $(T(e), !n)$, where e is a nonmapping representation type expression and $!n$ is a data representation variable for some numeral n.

An operator of an abstract data type is imported by specifying an import declaration containing the name of the operator followed by its representation type. For example, the operator *hd* of the abstract type *seq* might be imported by

import *hd* : *repty* (*seq* (⋆0), *File*) → ⋆0;

where *File* is a file representation of sequences. This import declaration implies that *hd* can only be applied to a sequence represented by *File*, a representation that may be unacceptable for some applications. A better import declaration for *hd* does not commit to a representation for *seq*, but rather lets a commitment come from an instantiation context. Such a declaration utilizes a data representation variable as in

import *hd* : *repty* (*seq* (⋆0), !1) → ⋆0;

Now the representation of *seq* is left unspecified as conveyed by the representation variable '!1' which gets bound in an instantiation context. In general, data representations are left unspecified because abstract data types can be implemented in many different ways. However, if a data representation is considered standard then its name may appear in an import declaration.

The predefined types are types for which one need never specify import declarations. Declarations for the operators of these types are provided by the software templates system which also declares representation types for integer, character and token constants. See Section 1 of the Appendix for system-supplied declarations.

***Definition* 2.1**. (type environment). A type environment is a set of pairs, each containing the name of an imported operator and its representation type.

2.2. Algorithm Specification

Algorithms are specified as typed first-order functions called templates. A template is recursively defined in terms of the operators of abstract data types and the conditional operator "$\rightarrow$;" which satisfies the equations

$$\mathit{true} \rightarrow x;\ y = x \qquad\qquad \mathit{false} \rightarrow x;\ y = y$$

For example, with the polymorphic sequence type *seq*, a template *rev* which reverses the elements of a sequence, can be defined recursively by

$$\mathit{rev} = \lambda x.\ \mathit{null}\,(x) \rightarrow \mathit{nil};\ \mathit{apndr}\,(\mathit{rev}\,(\mathit{tl}\ x), \mathit{hd}\ x)$$

It has type $\mathit{seq}\,(*) \rightarrow \mathit{seq}\,(*)$ which implies that it maps a sequence of objects of some type to a sequence of objects of the same type. Since *rev* does not commit to a type for sequence objects, it can be used to reverse sequences of objects of any type.

As another example, a template *assoc* which finds the value associated with a key in a sequence of key-value pairs, can be defined recursively by

$$assoc = \lambda(k, x).\ null\,(x) \rightarrow inr\,(\text{"}fail\ assoc\text{"});\\ eq\,(k, fst(hd\ x)) \rightarrow inl\,(snd\,(hd\ x));\ assoc\,(k, tl\ x)$$

The template *assoc*, a tail-recursive function defined in terms of predefined compound types, returns a value of union type to handle a missing key in which case an error message is generated. It has type

$$* \rightarrow (seq\,(pair\,(*, **)) \rightarrow union(**, tok))$$

The templates *rev* and *assoc* are examples of templates defined only in terms of abstract data type operators. In general, a template may be defined in terms of other functions. The template *rev* for instance, might be defined by

$$rev = \lambda x.\ null\,(x) \rightarrow nil;\ append\,(rev\,(tl\ x),\ apndl\,(hd\ x,\ nil))$$

where *append* is a function that appends two sequences. So in general, a template specification of an algorithm is not a single function definition as is the case for *rev* and *assoc*, but rather a family of typed, recursive function definitions.

There are two restrictions imposed on templates that distinguish them from functions of an applicative language like LML [4]. Both come from our intention to instantiate templates to components of imperative-language programs. 1) Every template is a first-order function, meaning it cannot take another function as input, or produce a function as its result. Most conventional imperative languages lack mechanisms such as procedures as first-class data objects [3], to represent higher-order functions. 2) Every operator (except "$\rightarrow$;") is interpreted strictly, meaning its operands are fully evaluated in every application.

3. SPECIFICATION OF ABSTRACT DATA TYPE IMPLEMENTATIONS

In the software templates methodology, implementations of abstract data types can be specified independently of algorithms. Specifying an implementation of an abstract data type involves declaring a data representation for values of the type, and implementing the operators of the type in terms of the declared representation. In a *complete* implementation, all operators including an assignment operator for variables of the type, are implemented. It is sometimes desirable however, to specify a *partial* implementation where only a subset of the operators is actually implemented. Partial implementations are permitted but they restrict the class of templates that can be instantiated. Like templates, implementations can be reused.

3.1. Data Representation

Unlike more traditional approaches to implementation [11, 12, 23], data representations in the templates paradigm can be declared independently of any abstract operator implementations. Data representations in essence serve as building blocks in representing abstract values. A data representation is declared by specifying a concrete type T_C of an imperative language, and a type declaration D_C that defines T_C. D_C is omitted if the concrete type T_C is predefined in the chosen imperative language. A declaration has the form

$$R() : T \quad [\, D_C \,] \; [\, T_C \,]$$

where R is a name for the data representation, and T is the abstract type whose values are being represented. In the representation R, we say that T_C represents T. The symbols ⟦ and ⟧, borrowed from denotational semantics, delimit text of the implementing imperative language. Text delimited by these symbols is not interpreted by the template instantiator. Consequently, an implementor is not confined to a single imperative language when implementing abstract data types. For example,

$$\textit{repchr}\,() : \textit{chr} \quad ⟦\texttt{char}⟧ \tag{3.1}$$

declares a data representation for the abstract type *chr*. In *repchr*, the concrete type `char` of the C programming language, represents *chr*. No type declaration is needed to define `char` because it is predefined in C [32]. As another example, consider

```
repseq() : chrseq
⟦struct cs {FILE *fp; char buf;};⟧
⟦struct cs⟧
```

which declares a data representation for the abstract type *chrseq* of Figure 2.1. In *repseq*, the type *chrseq* is represented by the concrete C type `struct cs`, which is defined by a type declaration since it is not predefined in C.

***Definition* 3.1.** (declared data representation). A declared data representation is a pair (D_C, T_C) where T_C is a concrete type defined by the type declaration D_C.

3.1.1. Polymorphism

The representations *repchr* and *repseq* represent the monomorphic types *chr* and *chrseq*. Without the capability to represent polymorphic types, monomorphic type representations proliferate, leading to unwieldy implementation libraries. For instance, for the sequence type alone there might be representations of $\textit{seq}\,(\textit{int})$,

seq (*chr*), *seq* (*seq* (*chr*)), and so on. This kind of proliferation can be avoided by specifying representations of polymorphic types such as *seq* (⋆).

A representation declared for a polymorphic type is called a *polymorphic data representation.* It is distinguished from a monomorphic representation by the use of a concrete type *scheme,* instances of which are created by the template instantiator to represent abstract values. If a concrete type T_C is defined by a type declaration containing a type variable (⋆0, ⋆1,...), then T_C is a concrete type scheme. This fact is conveyed to the instantiator by annotating T_C with a special prefix ($). For example, consider the declaration

```
useq : seq(*0)
[ struct $us {FILE *fp; *0 buf;}; ]
[ struct $us ]
```

The representation *useq* is polymorphic as conveyed by the concrete type scheme `struct $us`, which is defined in terms of the type variable `*0`. In *useq*, an instance of the concrete type scheme `struct $us` represents an instance of the abstract polymorphic type *seq* (⋆). The type variable `*0` gets bound to a representation type expression through a parameter-type commitment in an instantiation context. The type expression prescribes a concrete type which replaces `*0` in the type declaration during concrete code generation.

3.1.2. Parameterization

Data representations may be parameterized, affording users more control over the tailoring of a representation for a given application. Parameters get bound to values through an implementation commitment in an instantiation context. The

values replace the parameters when concrete code is generated. For example, the declaration

```
bseq(u) : seq(*0)
[ struct $bs {*0 elem[u]; int front,rear;}; ]
[ struct $bs ]
```

declares a data representation parameterized in array size. Whenever a user commits to *bseq* (selects it as a sequence representation in an instantiation context) an array size must be specified. The representation *bseq* exemplifies a bounded representation of sequences, unlike *useq* which is unbounded. In the templates paradigm, boundedness is viewed not as a property of abstract data types as in [22, 54], but rather as a property of representations [31].

3.2. Operator Implementation

Operators of an abstract data type are implemented in terms of declared data representations. An implementor is free to use either expressions or statements of an imperative language to implement an operator. These implementations must be free of side effects, however, incremental (in-place) updates are permitted for statement implementations. An operator is implemented by specifying a declaration of the form

$$pattern = [\ I\ ] \tag{3.2}$$

where I is an expression or a sequence of statements. Since the template instantiator is not allowed to interpret I, the pattern on the left-hand side conveys whether I is an expression or a statement. A pattern may contain one or more abstract variables, each of which an implementor chooses to represent in a particular way. The representations of these variables are used in the implementation I.

If I is an *expression* that implements an operator f then the pattern on the left-hand side of (3.2) has the form $f\ u_1, \ldots, u_n$ where $u_1, \ldots, u_n$ are abstract variables. In declaring these variables, an implementor chooses representations to be used in specifying I. For example, suppose w is an abstract variable declared by

$$w : useq;$$

In the scope of this declaration, any sequence to which w can be bound, is represented by *useq*. As a consequence of this declaration, an operator can be implemented in terms of the concrete type scheme `struct $us`, so long as the pattern used to declare the implementation contains the variable w. For example, the declaration

$$hd\ w = \llbracket\ w\texttt{.buf}\ \rrbracket \tag{3.3}$$

declares an expression implementation of the operator *hd* in terms of a component `buf` of the concrete type scheme `struct $us`. In the scope of the declaration "$w : bseq$;", another expression implementation of *hd* can be declared by

$$hd\ w = \llbracket\ w\texttt{.elem[}w\texttt{.front]}\ \rrbracket \tag{3.4}$$

If I is a sequence of *statements* that implements an operator f then the pattern appearing on the left-hand side of (3.2) either has the form $assign\,(v, f\ u_1, \ldots, u_n)$, or $assign\,(u_i, f\ u_1, \ldots, u_n)$ where $1 \leq i \leq n$. The latter pattern conveys an in-place update whereby I implements f by incrementally updating the value of the abstract variable u_i. Examples of these forms are given in Sections 3.2.1 and 3.2.2.

There is one additional form for the pattern in (3.2) that occurs when I is a sequence of statements that implements an *assignment* operator for variables of some abstract type. In this case, the pattern has the form $assign\,(v, u)$. For example, in

the scope of the declaration "*v*, *u* : *repchr*;", a C implementation of assignment for variables of type *chr* represented by *repchr* can be declared by

$$\textit{assign}\,(v,\,u) = [\![\, v \;=\; u\,; \,]\!] \tag{3.5}$$

By the declaration of *repchr*, the operands of the C assignment on the right-hand side have the concrete C type `char`. Usually an implementor has an interpretation of equality for abstract values expressed in terms of concrete types, from which an implementation of assignment can be inferred [17, 23].

***Definition* 3.2.** (declared implementation). A declared implementation is a pair (p, I) where I is an implementation of either an abstract operator or an assignment operator declared with the pattern p.

3.2.1. Polymorphism

As with data representations, implementing polymorphic operators is essential to prevent libraries from growing out of control with implementations of monomorphic operators. Polymorphic operators in general require implementations to be specified in terms of arguments whose types are unknown. For instance, an implementor should be able to specify an implementation of the polymorphic operator *apndr* even though the type of its second argument is unknown. To implement an operator such as this, an implementor specifies a *polymorphic implementation.*

A polymorphic implementation is characterized by a *scheme.* That is, I of (3.2) becomes a scheme, instances of which are created by the template instantiator to implement the operator specified in the pattern. I becomes a scheme if it is expressed in terms of a type variable (`*0`, `*1`,...), or an *abstract assignment.* A type variable,

because it ranges over representation types, can appear in *I* anywhere a concrete type might appear. For example, in the scope of the declaration "*w* : *useq*;", a statement implementation of *tl* expressed in C, can be declared by

assign (*w*, *tl w*) = ⟦ `fread(&w.buf,sizeof(*0),1,w.fp);` ⟧

The type variable `*0` appearing as the argument of the C operator `sizeof`, comes from the declaration of *useq*. It gets bound in an instantiation context to a representation type expression. The type expression prescribes a concrete type which replaces `*0` as the argument of `sizeof` during concrete code generation. The pattern appearing on the left-hand side of this implementation indicates that *tl* is implemented by an incremental update of the value bound to *w* (a file pointer is advanced).

3.2.1.1. Abstract Assignment

An abstract assignment enables an implementor to express assignment between two variables having a common but unknown type. Each such assignment is typed by a type variable which gets bound to a representation type in an instantiation context. If an implementor has provided an implementation of assignment for variables of this representation type, then it is used in the concrete code generation phase to expand the abstract assignment into a concrete one. For example, in the implementation of *cc* in Section 2.4 of the Appendix, there is an abstract assignment

```
assign((*_z.ctop->uval),_x,*0)
```

This abstract assignment is typed by the type variable `*0` which comes from the declaration of the representation *repcont*. Note that the abstract variable `_z` as conveyed by its declaration, is represented by *repcont*. As another example, consider the

pair constructor *pr* of the predefined compound type *pair*. Its standard C implementation as given in Section 2.3 of the Appendix, consists of two abstract assignments

```
assign(_y.left,_l,*0)
assign(_y.right,_r,*1)
```

The type variables `*0` and `*1` come from the data representation *reppair*. Abstract assignments may also appear in an implementation of assignment, as in the assignment implementation for variables of type *pair* in Section 2.3 of the Appendix.

3.2.2. Parameterization

Implementations of operators may be parameterized if they are specified in terms of representations with parameters. For example, a C implementation of *tl*

$$assign\,(w,\ tl\ w) = [\![\ w.\texttt{front} = (w.\texttt{front} + 1)\ \%\ \texttt{u};\ ]\!]$$

can be declared in the scope of the variable declaration "w : *bseq*;". The parameter `u`, from the declaration of *bseq*, gets bound in an instantiation context to some array size which replaces `u` in the right-hand side when concrete code is generated.

3.3. Representation Types

Since data types are abstract, it is natural to overload them with more than one implementation. The operator *hd* for instance, is overloaded with two expression implementations by the declarations (3.3) and (3.4). This overloading can be resolved by assigning to every declared implementation, a representation type inferred from a typing of the abstract variables appearing in the pattern and a type environment.

Every declared data representation has a type. Let T_C be a concrete type defined by a type declaration D_C. If in a representation named R, T_C represents the

abstract type T then the declared data representation (D_C, T_C) has type *repty* (T, R). For instance, the declared data representation (_, char) from (3.1) has type *repty* (*chr*, *repchr*), while the declared representation

```
(struct $us {FILE *fp; *0 buf;};, struct $us)
```

has type *repty* (*seq* (*0), *useq*). So abstract variables in essence get typed when they are declared. Using the types of abstract variables and a type environment, representation types are inferred for implementations, enabling them to be discriminated.

In a pattern used to declare an implementation of an operator, say *f*, the typing of abstract variables prescribes a representation type for *f*. If this type is β and *f* has type α in the type environment, then the implementation of *f* is assigned the inferred type $S[\alpha]$, where S is the most general unifier [51] of α and β. If there is no unifying substitution for α and β then the implementation declaration is ill-typed relative to the type environment. For example, with the type environment produced by the import declarations in Section 1.1 of the Appendix, the implementation declared by (3.3) is assigned the type *repty* (*seq* (*0), *useq*) → *0, while the implementation declared by (3.4) is assigned the type *repty* (*seq* (*0), *bseq* (u)) → *0.

Assignment too may become overloaded with implementations as new instances of assignment are implemented for variables of different abstract types. Like operators, overload can be resolved by ascribing a representation type to each assignment implementation. The type ascribed is simply obtained from the declarations of abstract variables. For instance, the assignment implementation declared by (3.5) is ascribed the type *repty* (*chr*, *repchr*) since the variables *u* and *v* are represented by the data representation *repchr*.

***Definition* 3.3.** (implementation environment). An implementation environment is a set of pairs, each containing either a declared data representation or a declared implementation, and its representation type.

3.4. Standard Implementation

The predefined compound types *union* and *pair*, and the type *cont* defined in Chapter 4, have standard implementations. That is, implementations of these types do not vary among instantiation contexts except across language boundaries. Standard C and Pascal implementations of the predefined compound types are given in Sections 2 and 3 of the Appendix.

3.5. Verification

An implementation of an abstract data type is said to be correct if corresponding abstract and concrete computations give corresponding results. Correspondence is more precisely defined by a representation function Φ which maps concrete values to abstract ones [27]. Let T_C be a concrete type that represents an abstract type T, so that Φ has type $T_C \rightarrow T$. If $f : T \rightarrow T$ is an abstract operator then the concrete operator $f_C : T_C \rightarrow T_C$ implements f if for all $x_C : T_C$, $\Phi(f_C(x_C)) = f(\Phi(x_C))$ [12].

A verification of an implementation is a proof that it is consistent with the axioms of the abstract type it implements [12, 23, 42]. In [23], an implementation is consistent if the left and right sides of every axiom can be reduced to identical terms using the implementation programs as rewrite rules. A system called "AFFIRM" has been developed to facilitate such consistency proofs [42]. In [12], verification conditions expressed in terms of an implementation, are formed from the axioms of an

abstract data type. If the conditions hold then the implementation is consistent with the axioms of the type. Unlike [23], abstract data types in [12] are implemented not in terms of rewrite rules, but rather in terms of an imperative language. Therefore, the templates methodology is more amenable to verification in the strategy of [12].

Although formal verification in the templates methodology is encouraged by the relatively small sizes of implementing code sequences, the complexity of proof systems for conventional, imperative languages may render it infeasible [1,45]. Therefore, we envision an investigation of consistency rather than a proof of it. A system called "DAISTS" (Data-Abstraction Implementation, Specification and Testing), has been developed to investigate consistency [17]. Axioms of an abstract data type drive implementing code on a collection of user-chosen test points. If there is a discrepancy between the left and right sides of an axiom then it is detected by the system.

4. TAIL-RECURSIVE TRANSFORMATION OF TEMPLATE SPECIFICATIONS

A template specification of an algorithm in general comprises a family of typed recursive function definitions. The first step of template instantiation removes recursion via a transformation. Each function definition is transformed into tail-recursive (iterative) form through the introduction of *continuations*. The result is a tail-recursive template, a family of typed tail-recursive function definitions.

4.1. Transformation Strategy

The transformation algorithm, attributed to Wand and Friedman [63], came about as a result of a study into the relationship between direct and continuation semantics. There are no restrictions on the kinds of recursion schemes (e.g. linear or dyadic) that can be transformed. However, the algorithm may produce tail-recursive functions whose evaluations still require an unbounded amount of space. This may even be true for functions that *can* be evaluated in bounded space, so in this sense, the algorithm is not optimal. Much of the effort spent in recursion removal is directed towards obtaining iterative computations that can be done in bounded space [2, 6, 46, 49, 58, 62]. The transformation algorithm presented here provides a good basis for developing recursion removal algorithms of this kind. Wand for instance, has described a recursion removal strategy based on finding closed forms for continuations [64]. Closed forms suggest efficient accumulator representations of continuations.

Before discussing the transformation strategy, some preliminary definitions and conventions must be given.

***Definition* 4.1.** (serious and trivial operators). A function defined in a template specification is a *serious* function or operator. The conditional operator "$\rightarrow$;" is a serious operator and all imported operators (Section 2.1.3) are *trivial.*

The names of serious operators are denoted by upper-case identifiers and those of trivial operators by lower-case identifiers. A term is a tree in which interior nodes are labeled with serious or trivial operators, and leaves are labeled with constants or variables. Any term free of nodes labeled with serious operators is a *trivial term.* Every term can be evaluated yielding a value whose type is the type of the term.

To motivate the use of continuations, an example is given in which the template *rev*, a linearly-recursive function, is transformed into a tail-recursive one by introducing a continuation. In the upper-case naming convention for serious operators, the template *rev* is defined by

$$REV = \lambda x.\ \mathit{null}(x) \rightarrow \mathit{nil};\ \mathit{apndr}(REV(\mathit{tl}\ x),\ \mathit{hd}\ x)$$

If REV is applied to a nonempty sequence x then evaluation proceeds by applying REV to $tl(x)$. If $REV(tl\ x)$ is the sequence v then REV applied to x is $apndr(v,\ hd\ x)$. Another way of expressing this evaluation is that the function $\lambda v.\ apndr(v,\ hd\ x)$ is applied to $REV(tl\ x)$. The function $\lambda v.\ apndr(v,\ hd\ x)$ is called a continuation [16, 26, 50, 55, 57]; it continues evaluation of $REV(x)$ following evaluation of $REV(tl\ x)$. A new function $REVC$ can be defined which when applied to a sequence and a continuation, reverses the sequence, and then applies the continuation to the result.

$$REVC = \lambda(x, \gamma).\ null(x) \rightarrow \gamma(nil);\ REVC(tl\ x, \lambda v.\ \gamma(apndr(v, hd\ x)))$$

Now $REV(x) = REVC(x, \lambda z.\ z)$ where $\lambda z.\ z$ is called the *null continuation.* Note that no function is called after the recursive call to *REVC.* Any function call with this property is called a *tail call.* If every call to a serious function in a definition is a tail call then the definition is tail-recursive. Thus, *REVC* is a tail-recursive function.

In general, the strategy for transforming a non-tail call C into a tail call, is to represent the computation that remains following C by a continuation, and then to replace the function in the call C by a new function for which the continuation is supplied as an argument. This new function is semantically equivalent to the function it replaces, however it takes as an additional argument, a continuation which it applies to the value it computes. For instance, *REVC* computes the reverse of a sequence and applies to it, the continuation bound to γ.

The tail-recursive transformation as described by Wand and Friedman [63], is influenced by a specific evaluation strategy for the transformed functions in which continuations are represented by data structures. Consequently, special functions must be defined to interpret these structures. In Section 4.2, the transformation is presented within a more general setting where continuations are treated as functions. In this setting, we can investigate properties of the transformation unencumbered by the details of any particular implementation. For the purpose of template instantiation however, this more general transformation must be specialized. In Section 4.4, a specialization is described that is based on a stack representation of continuations. Representing a continuation in this way entails stacking the values of all free variables in the continuation expression.

4.2. Transformation Rules

A function definition $F = \lambda(x_1, \ldots, x_n).\, e$ is transformed into tail-recursive form by replacing it with the definition

$$FC = \lambda(x_1, \ldots, x_n, \gamma).\, \delta[\gamma\ e]$$

The function FC computes F and applies to its result, the continuation bound to γ. δ is a transformation whose application (denoted by [ ]) to a continuation expression and a term, produces a *tail-call* term.

***Definition* 4.2.** (TC (tail-call) term). If γ is a continuation variable and $t_1, \ldots, t_n$ are trivial terms then $F(t_1, \ldots, t_n, \gamma)$ is a TC term. If t is a TC term then so is $F(t_1, \ldots, t_n, \lambda v.\ t)$. If p is a trivial term, and u_1 and u_2 are TC terms, then $p \rightarrow u_1;\ u_2$ is a TC term. Both γ and $\lambda v.\ t$ are called *continuation expressions.*

There are four rules that define the transformation δ. If t is a trivial term and K is a continuation expression then $K(t)$ is not a TC term. Suppose *SEND* is a function satisfying the equation $SEND(x, \gamma) = \gamma(x)$ for all x and any continuation γ. Then $K(t)$ can be rewritten as the TC term $SEND(t, K)$. So the first rule is

T1. $\delta[K\ t] \Rightarrow SEND(t, K)$

If $t_1, \ldots, t_n$ are trivial terms and K is a continuation expression then $K(F(t_1, \ldots, t_n))$ is not a TC term. This term can be rewritten as the TC term $FC(t_1, \ldots, t_n, K)$ if for all $x_1, \ldots, x_n$ and any continuation γ, FC is a function satisfying $FC(x_1, \ldots, x_n, \gamma) = \gamma(F(x_1, \ldots, x_n))$. So the second rule is

T2. $\delta[K\ F(t_1, \ldots, t_n)] \Rightarrow FC(t_1, \ldots, t_n, K)$

For a conditional to be a TC term, both branches must be TC terms. So δ must distribute over "$\rightarrow$;". If p is a trivial term, t_1 and t_2 are terms, and K is a continuation expression, then the third transformation rule becomes

T3. $\delta[K\ (p \rightarrow t_1;\ t_2)] \Rightarrow p \rightarrow \delta[K\ t_1];\ \delta[K\ t_2]$

If rules T1-T3 cannot be applied to a term then the term must be *factored.*

***Definition* 4.3**. (factorization). Let t be a term whose root is labeled with an operator f (either serious or trivial). A *factorization* of t is a triple (t', v, σ) where σ is a term substitution with three properties. 1) $vars\,(\sigma) = \{ v \}$ and v does not occur free in t. 2) $\sigma[v]$ is a proper subterm of t whose root is labeled with a serious operator, and such that f is strict in $\sigma[v]$. 3) t' is the term formed from t by replacing $\sigma[v]$ with v as illustrated in Figure 4.1 (the reason for introducing a term substitution will become clear when continuations are represented by a stack).

The strictness property of a factorization is imposed because the term $\sigma[v]$ will be evaluated before the term t'. Therefore, it must be the case that $\sigma[v]$ is

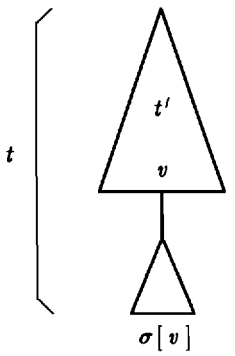

Figure 4.1. Factorization of a term t

evaluated unconditionally in the term t. A conditional $p \rightarrow t_1; t_2$ for instance, is a term in which the operator "$\rightarrow$;" may not be strict in a proper subterm of either t_1 or t_2, unless the subterm occurs in p.

If (t', v, σ) is a factorization of a term t, and K is a continuation expression, then the fourth transformation rule becomes

T4. $\delta[K\ t] \Rightarrow \delta[(\lambda v.\ \delta[K\ t'])\ \ \sigma[v]]$

As an example of the transformation δ, *REV* is transformed into tail-recursive form. The definition of *REV* is replaced by $REVC = \lambda(x, \gamma).\ e$ where e is defined by

$\delta[\gamma\ null(x) \rightarrow nil;\ apndr(REV(tl\ x), hd\ x)]$

$\Rightarrow null(x) \rightarrow \delta[\gamma\ nil];\ \delta[\gamma\ \ apndr(REV(tl\ x), hd\ x)]$ (T3)

$\Rightarrow null(x) \rightarrow SEND(nil, \gamma);\ \delta[(\lambda v.\ \delta[\gamma\ \ apndr(v, hd\ x)])\ \ REV(tl\ x)]$ (T1, T4)

$\Rightarrow null(x) \rightarrow SEND(nil, \gamma);\ REVC(tl\ x, \lambda v.\ \delta[\gamma\ \ apndr(v, hd\ x)])$ (T2)

$\Rightarrow null(x) \rightarrow SEND(nil, \gamma);\ REVC(tl\ x, \lambda v.\ SEND(apndr(v, hd\ x), \gamma))$ (T1)

$= null(x) \rightarrow \gamma(nil);\ REVC(tl\ x, \lambda v.\ \gamma(apndr(v, hd\ x)))$ (def. of SEND)

4.3. Transformation Properties

Transforming a term into a TC term proceeds by a sequence of rewrites with each application of δ bringing the term closer to TC form. The extent to which a term remains to be transformed can be measured by defining the *rank* of a term. It is basically a count of the number of proper subterms whose roots are labeled with serious operators.

$rank(t) = 0$ if t is trivial

$rank(F(t_1, \ldots, t_n)) = S(t_1) + \cdots + S(t_n)$

$rank(p \rightarrow t_1; t_2) = S(p) + rank(t_1) + rank(t_2)$

$S(t) = 0$ if t is trivial

$S(F(t_1, \ldots, t_n)) = 1 + S(t_1) + \cdots + S(t_n)$

$S(p \rightarrow t_1; t_2) = 1 + S(p) + rank(t_1) + rank(t_2)$

Each application of rule T3 or T4 reduces the rank of a term. Eventually, a term will be reached for which only T1 or T2 can be applied thereafter. Since no new instances of δ are introduced by either of the rules T1 or T2, the transformation will terminate.

The TC term derived by an application of δ need not be unique. In general, a factorization of a term is not unique and by computing different factorizations, more than one distinct TC term can be derived from a single starting term. For example, $H = \lambda x.\ F(G\ x, E\ x)$ is transformed into $HC = \lambda(x, \gamma).\ \delta[\gamma\ \ F(G\ x, E\ x)]$ where

$$
\begin{aligned}
&\delta[\gamma\ \ F(G\ x, E\ x)] \\
&\Rightarrow \delta[(\lambda v.\ \delta[\gamma\ \ F(v, E\ x)])\ \ G\ x] && \text{(T4)} \\
&\Rightarrow GC(x, \lambda v.\ \delta[\gamma\ \ F(v, E\ x)]) && \text{(T2)} \\
&\Rightarrow GC(x, \lambda v.\ \delta[(\lambda v'.\delta[\gamma\ \ F(v, v')])\ \ E\ x]) && \text{(T4)} \\
&\Rightarrow GC(x, \lambda v.\ EC(x, \lambda v'.\delta[\gamma\ \ F(v, v')])) && \text{(T2)} \\
&\Rightarrow GC(x, \lambda v.\ EC(x, \lambda v'.\ SEND(F(v, v'), \gamma))) && \text{(T1)}
\end{aligned}
$$

if $\sigma[v] = G(x)$ in the first application of rule T4. But, if $\sigma[v] = E(x)$ in the first application of rule T4, then a similar derivation results in the TC term

$$EC(x, \lambda v.\ GC(x, \lambda v'.\ SEND(F(v', v), \gamma)))$$

Thus starting with the same term $F(G\ x, E\ x)$, two distinct TC terms can be derived, so δ is not confluent as in [29, 44, 52].

In the current version of the instantiator, leftmost-outermost factorizations are computed. That is, $\sigma[v]$ is always the leftmost-outermost proper subterm whose root is labeled with a serious operator. The leftmost-outermost strategy has the desirable

property that whenever a conditional is factored, it is strict in $\sigma[v]$. In the preceding example, the instantiator would proceed according to the first derivation.

The following theorem establishes completeness for the transformation δ. That is, when δ terminates, it will have derived a TC term.

***Theorem* 4.1**. (completeness). If T is a term, K is a continuation expression and $\delta[K\ T]$ derives W upon termination, then W is a TC term.

Proof. The proof proceeds by structural induction on terms. The basis, terms free of "$\rightarrow$;", is established by an induction on the number of applications of rule T4 in the derivation of W from T.

Basis. Suppose T is a term free of "$\rightarrow$;". If W is derived from T by no applications of T4 then by rule T1, W has the form $SEND(T, K)$, or by rule T2, it has the form $FC(t_1, \ldots, t_n, K)$, and is therefore a TC term. For the inductive step, suppose (t', v, σ) is a factorization of the last term derived before the term W. Since $rank(t') < rank(t)$, a TC term for t' can be derived from T in fewer applications of T4 than are needed to derive W. So $\delta[K\ t']$ is a TC term by the inductive hypothesis. Since δ terminates with W, $\sigma[v]$ must be a term of the form $F(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are trivial terms. By rule T2, W has the form $FC(t_1, \ldots, t_n, \lambda v.\ \delta[K\ t'])$, and is therefore a TC term.

Inductive Step. Suppose $\delta[K\ t]$ is a TC term for any term t, and T is a term of the form $p \rightarrow t_1; t_2$ for terms p, t_1, and t_2.

Case 1. p is trivial. By rule T3, W has the form $p \rightarrow \delta[K\ t_1]; \delta[K\ t_2]$. By the inductive hypothesis, both branches are TC terms and therefore W is a TC term.

Case 2. p is not trivial. Let (t', v, σ) be a factorization of T such that t' is a term $p' \rightarrow t_1; t_2$ and p' is trivial. Then by Case 1, $\delta[K\ (p' \rightarrow t_1; t_2)]$ is a TC term. By rule T4, W has the form $\delta[(\lambda v.\ \delta[K\ (p' \rightarrow t_1; t_2)])\ \sigma[v]]$, and is therefore a TC term by the inductive hypothesis. □

4.4. Specialized Transformation

The transformation δ must be specialized for the purpose of template instantiation. As it stands, δ transforms functions into a continuation-passing style where continuations themselves are functions. Therefore, the functions generated by δ are unacceptable since they are higher-order functions. Thus, a data-structure representation of continuations is needed which will retain the expressiveness of continuations as functions, but permit definitions to be first-order. Continuations suggest a very natural representation, one usually found in conventional runtime environments, a stack. The remainder of this chapter describes a specialization of δ that is based on a stack representation of continuations.

4.4.1. Stack Representation of Continuations

The value of a continuation expression $\lambda v.\ e$ is a function or closure [64] formed by replacing each free variable of e by the value it had at the time the expression was evaluated.[1] In our representation, the values to which these free variables are bound are stacked. However, stacking is not restricted to the values of free variables only. Free variables may form the operands of trivial applications whose values can be stacked instead. For example, suppose x is bound to the sequence $<a, b>$. In

[1] Known in the LISP literature as static or proper binding [41].

representing the value of the continuation expression $\lambda v.\ \gamma(apndr\,(v,\ hd\ x))$ for some continuation γ, either $hd\,(x)$, or the entire sequence x can be stacked. Stacking the former in effect represents the continuation $\lambda v.\ \gamma(apndr\,(v,\ a))$, while stacking the latter represents

$$\lambda v.\ \gamma(apndr\,(v,\ hd\ <a,\ b>))$$

To represent a continuation, a set of trivial terms must be constructed whose values can be stacked. These trivial terms are called *closure terms.*

***Definition* 4.4.** (closure term). If t is a term whose root is labeled with an operator f (either serious or trivial), and t' is a trivial subterm of t such that f is strict in t', then t' is a *closure term* of t.

As we shall see, closure terms make up a special kind of factorization of a term and therefore must be evaluated unconditionally in the term. The values of closure terms are stacked by evaluating an application of cc which is a constructor of the polymorphic data type $cont\,(*)$ defined in Figure 4.2. Each application of cc creates a stack frame containing values of closure terms, and a label indicating where computation is to resume with these stacked values. Application of the constructor c of the

---

$cont\,(*) = quit\ +\ c\,(int,\ cont\,(*))\ +\ cc\,(int,\ *,\ cont\,(*))$
$isc\,(i,\ c\,(i,\ \_)) = true;$
$isc\,(i,\ cc\,(i,\ \_,\ \_)) = true;$
$tlc\,(c\,(\_,\ r)) = r;$
$tlc\,(cc\,(\_,\ \_,\ r)) = r;$
$val\,(cc\,(\_,\ v,\ \_)) = v;$

Figure 4.2. Polymorphic continuation data type

---

type *cont* (⋆) also creates a stack frame, but here the frame contains only a label. Labels in both *c* and *cc* constructions serve the same purpose as return addresses in a conventional runtime environment.

The label in the top stack frame is inspectable by *isc* and the closure-term values stacked in this frame can be extracted with *val*. The operator *tlc* is a stack pop operation, producing the tail of a sequence of stack frames. The nullary constructor *quit* is a special stack frame which signifies the end of a computation. Like the predefined compound types (Section 2.1.2), the abstract data type *cont* (⋆) has a standard implementation; see Sections 2.4 and 3.4 of the Appendix. A stack-frame sequence is the result of evaluating a *stack-frame expression.*

***Definition* 4.5.** (SF (stack-frame) expression). The constant *quit* is an SF expression and so is any variable ranging over stack-frame sequences. If l is a label and k is an SF expression then so is $c\,(l,\, k)$. If t is a trivial term then $cc\,(l,\, t,\, k)$ is an SF expression.

For example, if x is a trivial term then $cc\,(1,\, x,\, c\,(2,\, quit))$ is a stack-frame expression. Evaluation of this expression creates a stack-frame sequence comprised of two frames with the top frame being created by *cc*.

The stack representation only affects rule T4 which is defined by

$$\delta[K\ t] \Rightarrow \delta[(\lambda v.\ \delta[K\ t'])\ \ \sigma[v]]$$

The construction $\lambda v.\ \delta[K\ t']$ must be replaced by one that produces an SF expression. When evaluated, this SF expression must create a stack frame containing the values of any closure terms identified in a *closure-term* factorization of t, and a label that identifies a function whose application continues evaluation of t'.

***Definition* 4.6.** (CT (closure-term) factorization). A CT factorization of a term t is a factorization (t', v, σ) of the term t such that $vars(\sigma) = \{v, b_1, \ldots, b_n\}$, $\sigma[b_1], \ldots, \sigma[b_n]$ are closure terms, and t' is formed from t by replacing $\sigma[v]$ with v, and $\sigma[b_i]$ with b_i as illustrated in Figure 4.3.

The form of SF expression that gets produced in place of the continuation expression $\lambda v.\ \delta[K\ t']$ is contingent upon the CT factorization. If there are no closure terms then an SF expression must be produced which stacks only a label when evaluated. However, if there are closure terms then they must be packaged into a single term whose value can be stacked. Packaging multiple closure terms is accomplished by forming a pair expression. If $t_1, \ldots, t_n$ are closure terms then they are packaged by the pair expression $pr(t_1 \cdots pr(t_{n-1}, t_n) \cdots)$. To ensure that all stack frames have the same type, a value must be injected into a component of a union before it can be stacked. For each factorization that calls for a value to be

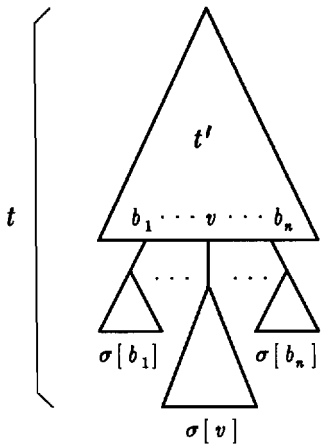

Figure 4.3. CT factorization of a term t

stacked, a new component type is added to the union. If there are a total of k such CT factorizations then every stack frame has the type $cont(U)$ where *cont* is the continuation data type, $U = union(T_1 \cdots union(T_{k-1}, T_k) \cdots)$, and T_i is the type of the value stacked in the i^{th} CT factorization.

4.4.2. Specialized Transformation Rules

Let Δ be a set of function definitions initialized as follows. For each function definition $F = \lambda(x_1, \ldots, x_n).\, e$ in a template specification, add to Δ, the definition

$$FC = \lambda(x_1, \ldots, x_n, s).\, \delta'[s\ e]$$

where s is a stack-frame variable, and δ' is a specialization of δ that is based on the stack representation of continuations. The rules of δ' are similar to δ except for rule T4. The first three rules of δ' are defined by

T1′. $\delta'[k\ t] \Rightarrow SEND(t, k)$
T2′. $\delta'[k\ F(t_1, \ldots, t_n)] \Rightarrow FC(t_1, \ldots, t_n, k)$
T3′. $\delta'[k\ (p \rightarrow t_1; t_2)] \Rightarrow p \rightarrow \delta'[k\ t_1]; \delta'[k\ t_2]$

where k ranges over SF expressions. Application of the fourth and final rule T4′ involves computing a CT factorization and constructing the expressions needed to stack the values of any closure terms.

Three auxiliary transformations are defined for rule T4′. The transformation *select* builds an expression to extract the value of a closure term in a pair.

$$select_{j,k}[t] \Rightarrow \textbf{if } j = k \textbf{ then } snd^{j-1}(t) \textbf{ else } fst(snd^{j-1}(t))$$

If the value of a term t is a k-component cascaded pair then $select_{j,k}[t]$ is an expression that evaluates to the j^{th} component. The transformations *in* and *out* build

expressions for injection into, and projection onto a union component.

$$in_{j,k}[t] \Rightarrow \textbf{if } j = k \textbf{ then } inr^{j-1}(t) \textbf{ else } inr^{j-1}(inl(t))$$
$$out_{j,k}[t] \Rightarrow \textbf{if } j = k \textbf{ then } outr^{j-1}(t) \textbf{ else } outl(outr^{j-1}(t))$$

The value of a term t is injected into the j^{th} component of a k-component union by evaluating the expression $in_{j,k}[t]$. If the value of a term t is a k-component union then $out_{j,k}[t]$ is an expression that evaluates to the j^{th} component.

Let (t', v, σ) be a CT factorization of a term t. If the factorization calls for a value to be stacked then suppose it is the m^{th} such factorization for Δ. If l is a unique label then rule T4′ adds to Δ, the new function definition

$$SEND_l = \lambda(v, s).\ (\lambda x.\ \delta'[x\ t']\ (tlc\ s)) \tag{4.1}$$

$SEND_l$ is applied to a value and a stack-frame sequence. Application of *tlc* pops the top stack frame revealing a new frame whose label is used to resume computation after t'. If s is a stack-frame sequence such that $isc(l, s)$ is true, then in order to apply the continuation represented by s to a value v, the function $SEND_l$ must be applied to v and s. Rule T4′ transforms the term t according to one of three cases.

Case I. No closure terms.

T4′.1. $\delta'[k\ t] \Rightarrow \delta'[c(l, k)\ \sigma[v]]$

Case II. One closure term $\sigma[b]$.

T4′.2. $\delta'[k\ t] \Rightarrow \delta'[cc(l, in_m[\sigma[b]], k)\ \sigma[v]]$

To extract the value of $\sigma[b]$ in t', b must be replaced in t' by the expression $out_m[val(s)]$ where s is the stack-frame variable of the new function $SEND_l$ in (4.1). Neither *in* nor *out* can be applied at this stage of the transformation since the total

number of CT factorizations calling for values to be stacked is not yet known.

Case III. n closure terms $\sigma[b_1], \ldots, \sigma[b_n]$.

T4′.3. $\delta'[k\ t] \Rightarrow \delta'[cc\,(l,\ in_m\,[p\,],\ k)\ \ \sigma[v\,]]$

where p is the pair expression $pr\,(\sigma[b_1] \cdots pr(\sigma[b_{n-1}], \sigma[b_n]) \cdots)$. To extract the value of the term $\sigma[b_j]$ in t', the variable b_j must be replaced in t' by the expression $select_{j,\,n}\,[out_m\,[val\,(s)]]$ where s is the stack-frame variable of the new function $SEND_l$ in (4.1).

When there are no longer any definitions in Δ remaining to be transformed, the total number of CT factorizations that resulted in stacked values can be supplied to every instance of *in* and *out*, thereby enabling injection and projection expressions to be constructed. However, definitions in Δ are still unacceptable since they contain instances of *SEND* that must be replaced by discriminating calls to the new functions introduced by rule T4′. For example, applying δ' to the template *rev* produces the tail-recursive template Δ_{rev} of Figure 4.4. Note that instances of *SEND* appear in the definitions of *REVC* and $SEND_1$, but *SEND* is not defined. Both instances must be replaced by a discriminating call to the functions $SEND_{quit}$ and $SEND_1$.

$$REVC = \lambda(x_1,\ x_0).\ null\,(x_1) \rightarrow SEND\,(nil,\ x_0);\ REVC\,(tl\ x_1,\ cc\,(1,\ hd\ x_1,\ x_0))$$

$$SEND_{quit} = \lambda(x_2,\ x_0).\ x_2$$

$$SEND_1 = \lambda(x_3,\ x_0).\ SEND\,(apndr\,(x_3,\ val\ x_0),\ tlc\ x_0)$$

Figure 4.4. Δ_{rev} before SEND discrimination

4.4.3. SEND Discrimination

By rule T1′, every call to *SEND* has the form *SEND* (t, k) where t is a trivial term and k is an SF expression. Each such call must be replaced by a call to a discriminating function, one that inspects the label in the top stack frame produced by an evaluation of k, and applies the function specified by the label. If k is an SF expression of the form $c(l, \_)$ or $cc(l, \_, \_)$ then after every evaluation of k, the top stack frame contains the label l, so *SEND* can be simply replaced by a call to $SEND_l$. If k is a stack-frame variable s then it may be necessary to replace *SEND* by a call to a function that discriminates labels. If rule T4′ was applied m times then there are $m + 1$ labels corresponding to $SEND_{quit}$, $SEND_{l_1}$, . . . , $SEND_{l_m}$. Replacing *SEND* by a call to a function that discriminates all labels may be unnecessary, since it may be an instance for which only a subset need be discriminated. Therefore, we determine the set of all labels that can appear in top stack frames produced by evaluations of s. It involves a form of abstract interpretation of definitions in Δ.

***Definition* 4.7.** (follows). If for a stack-frame sequence s, $isc(m, s)$ and $isc(n, tlc(s))$ are true, then label n is said to *follow* label m.

***Definition* 4.8.** (label set). If a label n follows a label m then n is in the *label set* of $SEND_m$ (denoted $ls(SEND_m)$). If the stack-frame variable of a function FC can become bound to a stack-frame sequence s in an application such that $isc(l, s)$ is true, then $l \in ls(FC)$.

A function μ is defined which computes for Δ, a set of rules M_Δ. Each rule prescribes an update of the label set for some function in Δ. When the label sets of all functions no longer change under updates prescribed by M_Δ, then the resulting sets

are the desired label sets. That is, the fixedpoint of M_Δ is the desired collection of label sets.

Each rule of M_Δ has the form $(R \mid S)$ which means bind R to the union of the sets S and R. M_Δ is the union of the sets of rules derived by applying μ to every function definition in Δ. Let $H = \lambda(x_1, \ldots, x_n, s).\ e$ be any function in Δ except $SEND_{quit}$. Then $\mu(e)$ is defined by cases.

Case I. $\mu(p \rightarrow t_1;\ t_2) = \mu(t_1) \cup \mu(t_2)$

Case II. $\mu(FC(t_1, \ldots, t_n, k))$ is the collection of rules

$$(ls(FC) \mid \{l_1\}),\ (ls(SEND_{l_1}) \mid \{l_2\}),\ \ldots,\ (ls(SEND_{l_{n-1}}) \mid \{l_n\}),\ (ls(SEND_{l_n}) \mid ls(H))$$

if k is a nested SF expression in the labels $l_1, \ldots, l_n$. Otherwise, it is the single rule $(ls(FC) \mid ls(H))$.

Case III. $\mu(SEND(t, k))$ is the collection of rules

$$(ls(SEND_{l_1}) \mid \{l_2\}),\ \ldots,\ (ls(SEND_{l_{n-1}}) \mid \{l_n\}),\ (ls(SEND_{l_n}) \mid ls(H))$$

if k is a nested SF expression in the labels $l_1, \ldots, l_n$. Otherwise, it is empty.

To get M_Δ started, the label set of a top-level function ($REVC$ for Δ_{rev}) is initialized with the special label *quit*. If $S = M_\Delta(S)$ for some collection S of label sets, then S is the desired collection.

After construction of label sets, a call to *SEND* of the form $SEND(t, s)$ where s is a stack-frame variable, can be replaced by a call to a discriminating function. If a call of this form appears in the definition of a function whose label set contains a single label l, then it can be simply replaced by a call to $SEND_l$. Otherwise, it must be

replaced by a call to a function that discriminates those labels in the label set. For example, both *REVC* and $SEND_1$ of Figure 4.4 have the label set $\{quit, 1\}$. Therefore, the calls to *SEND* in both *REVC* and $SEND_1$ are replaced by a call to a discriminating function *CSEND* which inspects the label in the top stack frame and applies either $SEND_1$ or $SEND_{quit}$; see Figure 4.5.

$$REVC = \lambda(x_1, x_0).\ null\,(x_1) \rightarrow CSEND\,(nil, x_0);\ REVC\,(tl\ x_1, cc\,(1, hd\ x_1, x_0))$$

$$SEND_{quit} = \lambda(x_2, x_0).\ x_2$$

$$SEND_1 = \lambda(x_3, x_0).\ CSEND\,(apndr\,(x_3, val\ x_0), tlc\ x_0)$$

$$CSEND = \lambda(x_4, x_0).\ isc\,(1, x_0) \rightarrow SEND_1(x_4, x_0);\ SEND_{quit}\,(x_4, x_0)$$

Figure 4.5. Δ_{rev} after SEND discrimination

5. REPRESENTATION TYPE INFERENCE

A template can be instantiated to a program component for many different applications by merely establishing new instantiation contexts. Different contexts usually call for different implementations of a template's trivial operators. It therefore becomes necessary to identify, for every instance of a trivial operator in a template, the implementation commitments and parameter-type commitments made in a context that affect its implementation. These commitments are conveyed by a representation type expression inferred from the template and the context. Type inference is a powerful form of abstract interpretation [39][1] that traditionally, has been used in type checking with an inferred type merely serving to document type correctness. But, our need for type inference goes beyond just type checking. Representation type expressions inferred for instances of trivial operators capture the commitments that are needed to correctly implement them.

5.1. Representation Typing

We define for a tail-recursive template, a representation typing. It ascribes to every instance of a trivial operator and to every variable, a representation type expression. A representation typing is computed by first constructing an initial typing, and then refining it through type inference.

---

[1] The idea of type inference originated with Roger Hindley, a British logician, but Robin Milner is credited with its application to programming.

An initial typing is constructed from a type environment and an instantiation context. The type environment specifies initial representation type expressions for all trivial operators, and constants of primitive type occurring in a tail-recursive template. In an initial typing, every instance of a trivial operator, and every constant of primitive type is ascribed the representation type specified for it in the environment. It is important to note that if the environment specifies a type expression containing variables then these variables are uniquely renamed each time the expression is ascribed to an instance. This is necessary in order to correctly interpret polymorphic operators [33]. Every instance of the conditional operator "$\rightarrow$;" is ascribed the representation type expression

$$repty(bool, !0) \rightarrow (*0 \rightarrow (*0 \rightarrow *0))$$

This type expression is given in curried form which merely simplifies the inference rules and is not intended to imply that the conditional operator is higher-order. The conditional operator is treated as a polymorphic operator, so for each instance of it, the variables $*0$ and $!0$ in the expression are uniquely renamed.

For example, an initial typing of the tail-recursive template Δ_{rev} of Figure 4.5 is given in Figure 5.1. The typing is constructed from the type environment produced by the import declarations in Section 1 of the Appendix. The initial type expressions ascribed to trivial operators are identical up to variable renaming, to those specified in the import declarations.

An instantiation context specifies initial representation type expressions for one or more variables of a tail-recursive template. These variables are ascribed the types specified for them in the context. Each remaining variable is ascribed the type of a

Variables

x_0: *114
x_1, x_2: *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*)
x_3: *115
x_4: *116

Operators

REVC
null : *repty* (*seq* (*101), !201) → *repty* (*bool*, *repbool*)
nil : *repty* (*seq* (*102), !202)
hd : *repty* (*seq* (*103), !203) → *103
tl : *repty* (*seq* (*104), !204) → *repty* (*seq* (*104), !204)
cc : *repty* (*int*, !205) → (*105 →
(*repty* (*cont* (*105), *repcont*) → *repty* (*cont* (*105), *repcont*)))

$SEND_1$
apndr : *repty* (*seq* (*107), !207) → (*107 → *repty* (*seq* (*107), !207))
val : *repty* (*cont* (*108), *repcont*) → *108
tlc : *repty* (*cont* (*109), *repcont*) → *repty* (*cont* (*109), *repcont*)

CSEND
isc : *repty* (*int*, !112) → (*repty* (*cont* (*113), *repcont*) → *repty* (*bool*, *repbool*))

Figure 5.1. An initial typing of Δ_{rev}

unique type variable. For example, suppose the template *rev* is to be instantiated with the directive **assign** (s', *rev* s) in the context established by the declaration

repty (*seq* (**repty** (*chr*, *repchr*)), *useq*) s, s';

An instantiation directive establishes a correspondence between the variables of a tail-recursive template, and the variables declared in a context. In the tail-recursive template Δ_{rev}, variable x_1 of *REVC* corresponds to s, and variable x_2 of $SEND_{quit}$ corresponds to s' (the first argument of $SEND_{quit}$ always corresponds to the target variable of a directive). Therefore, x_1 and x_2 are each ascribed the type expression *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*), and the remaining variables as illustrated in Figure 5.1, are ascribed the types of unique type variables.

5.1.1. Type Inference

An initial representation typing is refined through type inference. With an initial typing, representation types of expressions are inferred from their form and the types of their subexpressions. By virtue of inferring types, variables of initially-ascribed type expressions get bound through unification, leading to more refined type expressions. For example, suppose that in the application $hd\,(x_1)$, the trivial operator *hd* is initially typed as $repty\,(seq\,(\star 0),\ !0) \rightarrow \star 0$. If x_1 is initially typed as

$$repty\,(seq\,(repty\,(chr,\ repchr)),\ useq)$$

then the type $repty\,(chr,\ repchr)$ can be inferred for the application if $\star 0$ is bound to $repty\,(chr,\ repchr)$, and $!0$ is bound to *useq*. With these variables bound, the type expression ascribed to *hd* becomes more refined. When inferring types, variable bindings such as these are recorded in a type substitution.

***Definition* 5.1.** (type substitution). A type substitution is a substitution that maps type variables ($\star 0$, $\star 1$,...) to representation type expressions, and representation variables ($!0$, $!1$,...) to representations.

There is a type substitution I which behaves as the identity for composition of type substitutions. It is called the empty substitution and it replaces no variables. Type variables and representation variables get bound during an inference, through unification [14, 47, 51] of two type expressions. Unification is performed relative to a type substitution, producing perhaps a more refined type substitution, one that replaces new variables. Let $unify\,(\alpha,\ \beta,\ S)$ denote a type substitution identical to the substitution S except that it is updated by a most general unifier of the type expressions α and β relative to S.

A type inference function τ is defined which infers a representation type for a function application, a lambda abstraction, or a function definition. It is applied to each function definition of a tail-recursive template, returning for each definition a pair consisting of an inferred type expression and a type substitution. Before defining τ, we introduce notation for ascribed types.

***Definition* 5.2.** $(:_T)$. If T is a representation typing of a tail-recursive template then $f :_T$ denotes the representation type ascribed by T to an instance of f in the template. Likewise for variables, $x :_T$ denotes the representation type that T ascribes to the variable x of the template. We omit the subscript T whenever it is clear from the context of a discussion which typing is intended.

If RT is an initial typing then τ is defined by the following rules.

Rule I. (constant). If c is a constant of primitive type then $\tau(c) = (c :_{RT}, I)$.

Rule II. (variable). If x is a variable then $\tau(x) = (x :_{RT}, I)$.

Rule III. (application). If $f :_{RT} \alpha \rightarrow \beta$, $\tau(e) = (T, S')$ and $S = unify(\alpha, T, S')$ then for the application $f\ e$, $\tau(f\ e) = (S[\beta], S)$.

Rule IV. (abstraction). If $x :_{RT} \alpha$ and $\tau(e) = (T, S)$ then for the abstraction $\lambda x.\ e$, $\tau(\lambda x.\ e) = (S[\alpha] \rightarrow T, S)$.

Rule V. (definition). If $f :_{RT} \alpha$, $\tau(e) = (T, S')$ and $S = unify(\alpha, T, S')$ then for the definition $f = e$, $\tau(f = e) = (S[\alpha], S)$.

When applied to either a constant or a variable, τ returns an initially-ascribed type expression and the empty substitution.

The refinement of an initial typing can fail. If τ by Rule III or Rule V, should fail to produce a unifying substitution then type inference fails and the initial typing

cannot be refined. Failure can come as the result of a semantically-incorrect template, or an instantiation context in which the declared type of a variable suggests a typing of a trivial operator that is inconsistent with its type in the type environment.

As an example of type inference, the initial typing of Δ_{rev} in Figure 5.1 is refined by applying τ to each of the four function definitions of Δ_{rev}. Applying τ to the definition of *REVC* alone, refines its typing as illustrated in Figure 5.2. Note the variables *102 and !202 in the type expression ascribed to *nil*. Application of τ to *REVC* alone has not refined this expression. However, it will be refined once τ is subsequently applied to the remaining definitions. Applying τ to all definitions produces the representation typing of Figure 5.3, one for which no further refinement is possible. The type expression ascribed to *nil* is now refined in that *102 is bound to *repty* (*chr*, *repchr*), and !202 is bound to *useq*.

A change in the instantiation context for a template is represented by a new representation typing of its tail-recursive form. For instance, suppose that *rev* is to

---

Variables

x_0: *repty* (*cont* (*repty* (*chr*, *repchr*)), *repcont*)
x_1: *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*)

Operators

null : *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*) → *repty* (*bool*, *repbool*)
nil : *repty* (*seq* (*102), !202)
hd : *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*) → *repty* (*chr*, *repchr*)
tl : *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*) → *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*)
cc : *repty* (*int*, *repint*) → (*repty* (*chr*, *repchr*) →
(*repty* (*cont* (*repty* (*chr*, *repchr*)), *repcont*) → *repty* (*cont* (*repty* (*chr*, *repchr*)), *repcont*)))

Figure 5.2. A refined typing of *REVC*

be instantiated with the directive **assign** (s', *rev* s) in the context established by

repty (*seq* (**repty** (*chr*, *repchr*)), *bseq* (128)) s;
repty (*seq* (**repty** (*chr*, *repchr*)), *useq*) s';

The new context changes the initial typing of Δ_{rev} by altering the initial type ascribed to the variable x_1. The new representation typing of Δ_{rev} is given in Figure 5.4. Note that the representation types ascribed to *hd* and *tl* of *REVC* have changed (the only changes in the typing of trivial operators caused by the new context). The declarations establishing the new context illustrate implementation mixing [56] in the templates paradigm; two different sequence representations (*bseq* and *useq*) are being demanded within a single context.

Variables

x_0: *repty* (*cont* (*repty* (*chr*, *repchr*)), *repcont*)
x_1, x_2, x_3, x_4: *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*)

Operators

REVC
null : *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*) → *repty* (*bool*, *repbool*)
nil : *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*)
hd : *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*) → *repty* (*chr*, *repchr*)
tl : *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*) → *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*)
cc : *repty* (*int*, *repint*) → *repty* (*chr*, *repchr*) →
(*repty* (*cont* (*repty* (*chr*, *repchr*)), *repcont*) → *repty* (*cont* (*repty* (*chr*, *repchr*)), *repcont*)))

$SEND_1$
apndr : *repty* (*seq* (*repty* (*chr*, *repchr*)), *useq*) → (*repty* (*chr*, *repchr*) →
repty (*seq* (*repty* (*chr*, *repchr*)), *useq*))
val : *repty* (*cont* (*repty* (*chr*, *repchr*)), *repcont*) → *repty* (*chr*, *repchr*)
tlc : *repty* (*cont* (*repty* (*chr*, *repchr*)), *repcont*) → *repty* (*cont* (*repty* (*chr*, *repchr*)), *repcont*)

CSEND
isc : *repty* (*int*, *repint*) → (*repty* (*cont* (*repty* (*chr*, *repchr*)), *repcont*) → *repty* (*bool*, *repbool*))

Figure 5.3. A representation typing of Δ_{rev}

Variables

x_0: $repty\,(cont\,(repty\,(chr,\ repchr)),\ repcont)$
x_1: $repty\,(seq\,(repty\,(chr,\ repchr)),\ bseq\,(128))$
x_2, x_3, x_4: $repty\,(seq\,(repty\,(chr,\ repchr)),\ useq)$

Operators

REVC
$null$: $repty\,(seq\,(repty\,(chr,\ repchr)),\ useq) \rightarrow repty\,(bool,\ repbool)$
nil : $repty\,(seq\,(repty\,(chr,\ repchr)),\ useq)$
hd : $repty\,(seq\,(repty\,(chr,\ repchr)),\ bseq\,(128)) \rightarrow repty\,(chr,\ repchr)$
tl : $repty\,(seq\,(repty\,(chr,\ repchr)),\ bseq\,(128)) \rightarrow repty\,(seq\,(repty\,(chr,\ repchr)),\ bseq\,(128))$
cc : $repty\,(int,\ repint) \rightarrow repty\,(chr,\ repchr) \rightarrow$
$(repty\,(cont\,(repty\,(chr,\ repchr)),\ repcont) \rightarrow repty\,(cont\,(repty\,(chr,\ repchr)),\ repcont)))$

$SEND_1$
$apndr$: $repty\,(seq\,(repty\,(chr,\ repchr)),\ useq) \rightarrow (repty\,(chr,\ repchr) \rightarrow$
$repty\,(seq\,(repty\,(chr,\ repchr)),\ useq))$
val : $repty\,(cont\,(repty\,(chr,\ repchr)),\ repcont) \rightarrow repty\,(chr,\ repchr)$
tlc : $repty\,(cont\,(repty\,(chr,\ repchr)),\ repcont) \rightarrow repty\,(cont\,(repty\,(chr,\ repchr)),\ repcont)$

CSEND
isc : $repty\,(int,\ repint) \rightarrow (repty\,(cont\,(repty\,(chr,\ repchr)),\ repcont) \rightarrow repty\,(bool,\ repbool))$

Figure 5.4. Another representation typing of Δ_{rev}

Type inference is very powerful, capable of handling contextual changes in either parameter types or implementations. Our examples have illustrated only changes in implementations, but changes in the parameter type of *seq* are also permitted since *seq* is polymorphic. Representation typings capture both kinds of contextual changes, reflecting any new commitments. As the contexts for a template change, only the typings of its tail-recursive form vary.

6. TRANSLATION INTO INTERMEDIATE IMPERATIVE-LANGUAGE CODE

An algorithm is presented for translating a representation-typed, tail-recursive template into intermediate imperative-language code called L code (Locations). L is a primitive imperative language providing only assignment and very basic control structures, making it amenable to translation into concrete code for a wide variety of imperative languages.

By virtue of a template being tail-recursive, every serious function call has the form $F(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are trivial terms. For every such call, a pair (S, P) is constructed where S and P are compositions of L assignment statements. Execution of S evaluates the actual parameters $t_1, \ldots, t_n$, and execution of P binds their values to the formal parameters of F. For a conditional $p \rightarrow t_1; t_2$, a composition of L assignment statements is constructed which evaluates the predicate p.

6.1. S Construction

The S component for a serious function call is computed through a *state-building* reduction of the call. A call is reduced by identifying a set of innermost operator applications which form the right sides of assignment statements, and replacing the applications in the call by the variables appearing on the left sides of these assignments. A call is repeatedly reduced until it reaches a *normal form* $F(v_1, \ldots, v_n)$ where $v_1, \ldots, v_n$ are variables. At this time, the S component for the call is formed

by composing the assignment sequences constructed over all reduction steps. The P component for the call is constructed from the normal form.

A function *COMPOSE* is defined which computes the S component for a serious function call. When applied to a serious call, an initial composition of assignments *skip*, and a term substitution, *COMPOSE* returns a normal form, the S component for the call, and an updated term substitution. A term substitution which maps variables to trivial terms, records those assignments that are collapsed during a translation. If a trivial operator f is implemented by an expression then an assignment $v \leftarrow f\ x$ is collapsed by updating a term substitution with the pair $(v, f(x))$. Before defining *COMPOSE*, some preliminary definitions must be given.

***Definition* 6.1.** (trivial application). A trivial application is a constant, a variable, or a trivial term of the form $f(v_1, \ldots, v_k)$ where $v_1, \ldots, v_k$ are variables. The set of trivial applications for a term t is given by $TA(t)$ where TA is defined by

$$TA(t) = \{t\} \text{ if } t \text{ is a trivial application}$$
$$TA(f(t_1, \ldots, t_n)) = TA(t_1) \cup \cdots \cup TA(t_n)$$

***Definition* 6.2.** ($<$). If $u \leftarrow t$ and $v \leftarrow t'$ are assignments for trivial terms t and t', and v occurs in t, then $u \leftarrow t$ stands in relation *fetch-store* to $v \leftarrow t'$ denoted $u \leftarrow t < v \leftarrow t'$ [34].

If *normal* is a boolean-valued function that tests for normal form then the function *COMPOSE* is defined by

$$\begin{aligned} COMPOSE = \lambda(t, S, \sigma).\ &\textbf{if}\ \ normal(t)\ \ \textbf{then}\ \ (t, S, \sigma) \\ &\textbf{else}\ \ \textbf{let}\ \ (t', S', \sigma') = REDUCE(t, \sigma)\ \ \textbf{in} \\ &\quad COMPOSE(t', S; S', \sigma') \end{aligned}$$

and *REDUCE* is defined by

$$\begin{aligned} \mathit{REDUCE} = \lambda(t, \sigma).\ & \textbf{let}\ (V, E) = \mathit{FETCH\text{-}STORE}(t)\ \textbf{in} \\ & \textbf{let}\ l = \mathit{LINEAR}\ ((V, E))\ \textbf{in} \\ & \textbf{let}\ (S, \sigma') = \mathit{COLLAPSE}\ ((V, E), l, \mathit{skip}, \sigma)\ \textbf{in} \\ & (\mathit{REWRITE}(t, V), S, \sigma') \end{aligned}$$

Let RT be a representation typing of a tail-recursive template, and let Λ be an implementation environment. Implementations of trivial operators are selected from Λ according to ascribed representation types. RT ascribes a representation type to every instance of a trivial operator, and to every variable of the tail-recursive template. If $f :_{RT} T$, for an instance of a trivial operator f, then the declared implementation of f whose representation type matches T is selected from Λ for this instance of f. If there is no such declared implementation then selection fails for Λ. Matching is a first-order instantiation where T is not allowed to contain any variables.[1] A match produces a type substitution called the *matching substitution.* The functions *FETCH-STORE, LINEAR, COLLAPSE,* and *REWRITE* are defined as follows.

FETCH-STORE (t). An assignment is added to an initially empty vertex set V for each nonvariable term $t' \in TA(t)$. If t' is a constant c of primitive type then add $v \leftarrow c$ to V where v is a new variable. Otherwise, t' is a term of the form $f(v_1, \ldots, v_k)$. Let p be the pattern of the declared implementation selected from Λ for this instance of f. There are two cases to consider in constructing an assignment.

Case I. p is a pattern $f\ u_1, \ldots, u_k$ or *assign* $(v, f\ u_1, \ldots, u_k)$. Add the assignment $v \leftarrow f\ v_1, \ldots, v_k$ to V where v is a new variable.

---

[1] If matching were a more general first-order unification then T could contain variables and might match the representation type of more than one implementation in Λ.

Case II. p is a pattern *assign* $(u_i, f\ u_1, \ldots, u_k)$. Add $v_i \leftarrow f\ v_1, \ldots, v_k$ to V.

Now for each variable $v \in TA\,(t)$ such that v is the target variable of an assignment in V, add $v' \leftarrow v$ to V where v' is a new variable. The value bound to v is incrementally updated but its original value must be retained, so a copy is made. An edge set E is constructed by adding an edge (a, b) if a and b are assignments in V such that $a < b$. Return the directed graph (V, E).

LINEAR $((V, E))$. If the relation represented by the directed graph (V, E) is a partial order then *LINEAR* returns a consistent linear order. The *fetch-store* relation constructed by *FETCH-STORE* may not be a partial order. Consider for example, a vertex set containing the assignments $u \leftarrow v$ and $v \leftarrow u$. If the relation represented by (V, E) is not a partial order then failure results for Λ.

COLLAPSE $((V, E), l, S, \sigma)$. Visit each vertex $v \leftarrow t \in V$ according to the linear order l starting with the least vertex. If t is a constant c then update the term substitution σ with the pair (v, c) (denoted $\sigma \mid v : c$). If t is a variable y then let S be S; $v \leftarrow y$ and $\sigma \mid v : v$. Otherwise, t is a term of the form $f\,(v_1, \ldots, v_k)$. Let p be the pattern of the declared implementation selected from Λ for this instance of f. If p is a pattern $f\ u_1, \ldots, u_k$ and $v \leftarrow t$ has no outgoing edges then $\sigma \mid v : t$. Otherwise, let S be S; $v \leftarrow \sigma[t]$ and $\sigma \mid v : v$. Return (S, σ).

REWRITE (t, V). For each assignment $v \leftarrow t' \in V$, replace all instances of t' in t by v. Return the new term.

A conditional $p \rightarrow t_1; t_2$ can be treated as a special kind of serious call, one for which there is no P component. Instead, a pair (S, v) is associated with the conditional where v is a variable and S is an assignment composition which when executed,

evaluates the predicate p. The S component is constructed by applying *COMPOSE* to the term $\rightarrow(p)$. If *COMPOSE* returns the normal form $\rightarrow(w)$ and the composition s then the S component of the pair is s, and the v component is w. Failure results for Λ if there is a composition (S or P) computed for either t_1 or t_2, containing an assignment that stands in relation *fetch-store* to an assignment in s.

Following computation of the S component for a serious call, a composition can be formed for the P component. If for a call $F(t_1, \ldots, t_n)$, the initial composition *skip*, and a term substitution σ,

$$COMPOSE(F(t_1, \ldots, t_n), skip, \sigma) = (F(v_1, \ldots, v_n), s, \sigma')$$

then a composition for the P component is given by

$$x_1 \leftarrow \sigma'[v_1]; \cdots; x_n \leftarrow \sigma'[v_n]$$

where $x_1, \ldots, x_n$ are the formal parameters of F. A composition for the P component may change as the result of a variable merge described in the next section. The merge may make some variables equivalent rendering some or even all assignments in the P component redundant in which case they can be eliminated.

6.2. P Construction

Constructing the P component for a call involves eliminating redundant assignments and building an execution schedule for those that remain. An assignment of the form $v \leftarrow u$ for variables v and u, is not redundant but can be made so if u and v are merged into a single variable. Therefore, a merge algorithm is employed to construct an equivalence relation on variables. If two variables u and v are equivalent in the constructed relation then the assignment $v \leftarrow u$ can be eliminated.

6.2.1. Variable Merge

The merge algorithm employed is an adaptation of the one described in [63]. Let σ be the final term substitution derived by applying *COMPOSE* to every serious function call of a tail-recursive template. The algorithm constructs an equivalence relation on the set $\{ v \mid \sigma[v] = v \}$, and calls for each equivalence class to have an associated set of users.

***Definition* 6.3.** (users). If x is a formal parameter of a serious function F, or a variable created by applying *COMPOSE* to the definition of F, then $F \in users([x])$.

The algorithm begins by creating an equivalence class $[v]$ for every variable v for which $\sigma[v] = v$. All P compositions are then examined for assignments of the form $v \leftarrow u$ where u and v are variables. An equivalence-class merge is then attempted for each of these assignments. For an assignment $v \leftarrow u$, $[v]$ and $[u]$ are merged if the $users([u])$ and $users([v])$ are disjoint sets.

6.2.2. Scheduling

Nonredundant assignments in every P composition must be scheduled for execution. Scheduling must be done with respect to the equivalence relation constructed by the variable merge algorithm. For example, there are no scheduling constraints on the execution of the assignments $v \leftarrow u$ and $y \leftarrow x$. But, if an equivalence relation is constructed in which v and x are equivalent with representative say w, then the only admissible composition is $y \leftarrow w$; $w \leftarrow u$.

Let R be an equivalence relation constructed by the merge algorithm, and let V be the set of assignments that remain in a P composition after assignments deemed

redundant under R, are eliminated. An edge set E is constructed by creating an edge (a, b) for every pair of assignments $a, b \in V$ such that $a <_R b$.

***Definition* 6.4.** $(<_C)$. Let t and t' be trivial terms in the assignments $u \leftarrow t$ and $v \leftarrow t'$. If $w\ C\ v$ and w occurs in t, then $u \leftarrow t <_C v \leftarrow t'$.

If the relation represented by the directed graph (V, E) is a partial order then it is embedded in a consistent linear order, otherwise failure results for Λ. A final P composition is formed by composing the assignments of V according to the linear order such that the least assignment is the first one executed.

6.3. L Code Generation

L code can be generated for a tail-recursive template after a composition pair is computed for every serious function call, and a composition-variable pair is computed for every conditional. First, L declarations are generated for the representative variables of the equivalence classes constructed by the variable merge algorithm. Code is then generated for each function by traversing its parse tree. When a call to a serious function F is encountered, its composition pair (S, P) is used to generate the code

$$S; P;\ \textbf{goto}\ F$$

When a conditional $p \rightarrow t_1; t_2$ is encountered, its composition-variable pair (S, v) is used to generate the code

$$S; \textbf{if}\ \sigma[v]\ \textbf{then}\ l_1\ \textbf{else}\ l_2$$

where σ is the final term substitution, and l_1 and l_2 are L code sequences generated for t_1 and t_2 respectively.

6.4. Example -- rev

An example of L code generation is given for the tail-recursive template Δ_{rev} of Figure 4.5. Let RT denote the representation typing of Figure 5.4, and let Λ be the implementation environment defined by the declared C implementations in Section 2 of the Appendix. A composition pair is computed for every serious function call, and a composition-variable pair is computed for the conditionals of *REVC* and *CSEND*.

A set of formal parameters is defined for each function of Δ_{rev} by uniquely renaming all variables. Let the formal parameters of each function be defined by

REVC	x_1, x_0
$SEND_{quit}$	x_2, x_5
$SEND_1$	x_3, x_6
CSEND	x_4, x_7

Let σ be a term substitution initialized so that $\sigma[v] = v$ for each formal parameter v. We begin by computing a composition pair for every serious function call starting with the definition of *REVC*. For the serious call $CSEND\,(nil, x_0)$,

$$\begin{aligned}
&COMPOSE\,(CSEND\,(nil, x_0), skip, \sigma)\\
&= COMPOSE\,(CSEND\,(y_0, x_0), skip;\ y_0 \leftarrow nil, \sigma \mid y_0 : y_0)\\
&= (CSEND\,(y_0, x_0), y_0 \leftarrow nil, \sigma \mid y_0 : y_0)
\end{aligned}$$

This triple is arrived at as follows. For this instance of *nil*,

$$nil :_{RT} repty\,(seq\,(repty\,(chr, repchr)), useq)$$

There is a declared implementation of *nil* in Λ whose representation type $repty\,(seq\,(*O), useq)$ matches the type ascribed to *nil* by RT. Therefore this implementation is selected and its pattern $assign\,(\_s, nil)$, is used by *FETCH-STORE* to create a vertex set consisting of the single assignment $y_0 \leftarrow nil$, where y_0 is a new

variable. The edge set is empty so *LINEAR* returns a trivial linear ordering of a single vertex. The assignment $y_0 \leftarrow nil$ cannot be collapsed so *COLLAPSE* returns the composition $skip; y_0 \leftarrow nil$ and the substitution $\sigma \mid y_0 : y_0$. Finally, *REWRITE* returns the normal form $CSEND\,(y_0, x_0)$, terminating *COMPOSE*. The P composition for this call is given by $x_4 \leftarrow \sigma[y_0]; x_7 \leftarrow \sigma[x_0]$. So the composition pair (S, P) for this serious call becomes

$$S = y_0 \leftarrow nil$$
$$P = x_4 \leftarrow y_0; x_7 \leftarrow x_0$$

Next, a composition pair is computed for the call $REVC\,(tl\ x_1, cc\,(1, hd\ x_1, x_0))$. A reduction sequence is given for the application of *COMPOSE* to this term. The new term and composition returned by *REDUCE* is specified for each reduction.

$$\begin{aligned} & REVC\,(tl\ x_1, cc\,(1, hd\ x_1, x_0)) \\ \Rightarrow\ & REVC\,(x_1, cc\,(y_2, y_1, x_0)) && y_1 \leftarrow hd\ x_1; x_1 \leftarrow tl\ x_1 \\ \Rightarrow\ & REVC\,(x_1, x_0) && x_0 \leftarrow cc\ 1\ y_1\ x_0 \end{aligned}$$

COMPOSE returns the substitution $(\sigma \mid y_1 : y_1) \mid y_2 : 1$. The composition pair is

$$S = y_1 \leftarrow hd\ x_1; x_1 \leftarrow tl\ x_1; x_0 \leftarrow cc\ 1\ y_1\ x_0$$
$$P = x_1 \leftarrow x_1; x_0 \leftarrow x_0$$

For the conditional of *REVC*, a composition-variable pair is computed.

$$\begin{aligned} & COMPOSE\,(\rightarrow(null\,(x_1)), skip, \sigma) \\ & = COMPOSE\,(\rightarrow(y_3), skip, \sigma \mid y_3 : null\,(x_1)) \\ & = (\rightarrow(y_3), skip, \sigma \mid y_3 : null\,(x_1)) \end{aligned}$$

The composition-variable pair is $(skip, y_3)$.

Composition pairs are now computed for the serious calls of $SEND_1$ and *CSEND*. For the serious call $CSEND\,(apndr\,(x_3,\ val\ x_6),\ tlc\ x_6)$,

$$\begin{aligned} & CSEND\,(apndr\,(x_3,\ val\ x_6),\ tlc\ x_6) \\ \Rightarrow\ & CSEND\,(apndr\,(x_3,\ y_4),\ x_6) && y_4 \leftarrow val\ x_6;\ x_6 \leftarrow tlc\ x_6 \\ \Rightarrow\ & CSEND\,(x_3,\ x_6) && x_3 \leftarrow apndr\ x_3\ y_4 \end{aligned}$$

COMPOSE returns the substitution $\sigma \mid y_4 : y_4$. The composition pair is

$$\begin{aligned} S &= y_4 \leftarrow val\ x_6;\ x_6 \leftarrow tlc\ x_6;\ x_3 \leftarrow apndr\ x_3\ y_4 \\ P &= x_4 \leftarrow x_3;\ x_7 \leftarrow x_6 \end{aligned}$$

For the serious call $SEND_1\,(x_4,\ x_7)$, the composition pair is

$$\begin{aligned} S &= skip \\ P &= x_3 \leftarrow x_4;\ x_6 \leftarrow x_7 \end{aligned}$$

and for the serious call $SEND_{quit}\,(x_4,\ x_7)$,

$$\begin{aligned} S &= skip \\ P &= x_2 \leftarrow x_4;\ x_5 \leftarrow x_7 \end{aligned}$$

Finally, for the conditional of *CSEND*,

$$\begin{aligned} & COMPOSE\,(\rightarrow(isc\,(1,\ x_7)),\ skip,\ \sigma) \\ &= COMPOSE\,(\rightarrow(isc\,(y_5,\ x_7)),\ skip,\ \sigma \mid y_5 : 1) \\ &= COMPOSE\,(\rightarrow(y_6),\ skip,\ (\sigma \mid y_6 : isc\,(y_5,\ x_7)) \mid y_5 : 1) \\ &= (\rightarrow(y_6),\ skip,\ (\sigma \mid y_6 : isc\,(y_5,\ x_7)) \mid y_5 : 1) \end{aligned}$$

The composition-variable pair is $(skip,\ y_6)$.

Now we attempt to make assignments in the P compositions redundant. The following is a list of the normal forms and P compositions they generated.

$CSEND\,(y_0,\ x_0)$	$x_4 \leftarrow y_0;\ x_7 \leftarrow x_0$
$REVC\,(x_1,\ x_0)$	$x_1 \leftarrow x_1;\ x_0 \leftarrow x_0$

$CSEND\,(x_3,\ x_6)$	$x_4 \leftarrow x_3;\ x_7 \leftarrow x_6$
$SEND_1\,(x_4,\ x_7)$	$x_3 \leftarrow x_4;\ x_6 \leftarrow x_7$
$SEND_{quit}\,(x_4,\ x_7)$	$x_2 \leftarrow x_4;\ x_5 \leftarrow x_7$

We begin by constructing an equivalence relation on those variables of σ for which $\sigma[v] = v$. This set is defined by $\{x_0, \ldots, x_7, y_0, y_1, y_4\}$. An equivalence class is created for each of these variables. There are six assignments for which equivalence-class merges are attempted. For the assignment $x_7 \leftarrow x_0$, $[x_7]$ can be merged with $[x_0]$ since $users\,([x_7])$ is the set $\{CSEND\}$, and $users\,([x_0])$ is the set $\{REVC\}$. The union of these two user sets is the user set of the new class $[x_0, x_7]$. The merge algorithm can be summarized by listing each assignment for which a merge is attempted, and the equivalence relation that results.

$x_7 \leftarrow x_0$	$[x_0, x_7], \ldots, [x_6], [y_0], [y_1], [y_4]$
$x_7 \leftarrow x_6$	$[x_0, x_7, x_6], \ldots, [x_5], [y_0], [y_1], [y_4]$
$x_5 \leftarrow x_7$	$[x_0, x_7, x_6, x_5], \ldots, [x_4], [y_0], [y_1], [y_4]$
$x_4 \leftarrow y_0$	$[x_0, x_7, x_6, x_5], \ldots, [x_4, y_0], [y_1], [y_4]$
$x_4 \leftarrow x_3$	$[x_0, x_7, x_6, x_5], \ldots, [x_4, y_0, x_3], [y_1], [y_4]$
$x_2 \leftarrow x_4$	$[x_0, x_7, x_6, x_5], [x_1], [x_4, y_0, x_3, x_2], [y_1], [y_4]$

The constructed equivalence relation is used to identify redundant assignments that can be eliminated in each P composition. Those assignments that remain must be scheduled for execution. However for this example, there are no P compositions with nonredundant assignments, so no scheduling is necessary.

L code can now be generated using the composition pairs computed for serious function calls. Let x_0 and x_2 be equivalence-class representatives. Declarations are generated for the representatives of the five equivalence classes. The L code generated for Δ_{rev} is given in Figure 6.1.

```
decl x0 repty(cont(repty(chr, repchr)), repcont)
decl x1 repty(seq(repty(chr, repchr)), bseq(128))
decl x2 repty(seq(repty(chr, repchr)), useq)
decl y1 repty(chr, repchr)
decl y4 repty(chr, repchr)

label REVC
if null(x1) then
        x2 ← nil; goto CSEND
else
        y1 ← hd x1;
        x1 ← tl x1;
        x0 ← cc 1 y1 x0;
        goto REVC
end

label SEND1
y4 ← val x0;
x0 ← tlc x0;
x2 ← apndr x2 y4;
goto CSEND

label CSEND
if isc(1, x0) then goto SEND1
else goto SENDquit
end

label SENDquit
```

Figure 6.1. Translation of Δ_{rev} into L code

6.5. Translation Properties

In forming P compositions, a variable merge algorithm is employed to build an equivalence relation on variables. A set containing assignments of the form $v \leftarrow u$ is identified, suggesting a set of variable pairs eligible for merging. How many of the pairs result in successful merges can depend on the order in which merges occur. For example, suppose (y, u), (v_1, u), (v_2, u), (x_1, y) and (x_2, y) are variable pairs eligible for merging. The initial collection of equivalence classes is formed by letting

$[v] = \{v\}$ for each variable v. Suppose the user sets are defined by

$$\textit{users}([u]) = \{F\} \qquad \textit{users}([x_1]) = \textit{users}([v_1]) = \{G\}$$
$$\textit{users}([y]) = \{M\} \qquad \textit{users}([x_2]) = \textit{users}([v_2]) = \{K\}$$

Merging the pairs (x_1, y), (x_2, y) and (y, u) prevents two eligible pairs (v_1, u) and (v_2, u) from being merged. The equivalence relation constructed by merging variable pairs in this order is given by the following equivalence classes.

$$[x_1, x_2, y, u]\ [v_1]\ [v_2]$$

Merging instead, the pairs (x_1, y), (x_2, y), (v_1, u) and (v_2, u) only prevents one eligible pair (y, u) from being merged. The equivalence relation now becomes

$$[x_1, x_2, y]\ [v_1, v_2, u]$$

The merge algorithm employed by the instantiator merges variables arbitrarily and may not construct the minimum number of equivalence classes. Therefore in this sense, the algorithm is not optimal. However, the problem of constructing the minimum number of classes is conjectured to be NP-hard [63]. Although the merge algorithm is not optimal, it will merge every eligible pair of variables whenever this is possible. Determining if every eligible pair of variables can be merged is not contingent upon merge order. If one merge order fails to merge every eligible pair then no merge order can result in all eligible pairs being merged.

***Theorem* 6.1**. (merge order). If one merge order fails to merge every eligible pair of variables then all merge orders fail to do so.

Proof. Let (u, v) be a pair whose merging prevents the pair (x, y) from being merged. That is, the merging of (u, v) forces the user sets of $[x]$ and $[y]$ to intersect.

Then $y \in [v]$ and the user sets of $[u]$ and $[x]$ intersect. But choosing to merge (x, y) before (u, v) prevents (u, v) from being merged since $v \in [x]$ and the user sets of $[u]$ and $[x]$ intersect. □

This result is significant when eliminating those assignments for which there are no declared implementations. Assignments of this kind will cause instantiation to fail unless they are eliminated. If the merge algorithm fails to merge those pairs which would render such assignments redundant then there is some comfort in knowing that no amount of backtracking among merge orders will succeed in merging them.

6.5.1. Correctness

In this section, *COMPOSE* is proved correct. The proof requires a formal semantics for assignment and composition in L. Such a semantics is given in Figure 6.2. We incorporate syntax into the syntactic domain definitions, as in the definition of *Exp*. The domain *Opr* represents trivial operators. Only a single domain of values is defined since there is no distinction between denotable and storable values. Constants are to be interpreted as nullary operators of the domain *Opr*. Their meaning is given by the semantic equation for operator application with $n = 0$. Informally, *COMPOSE* is proved correct by showing that the S composition it constructs to evaluate the actual parameters of a serious function call, preserves the functional semantics of the actual parameters.

***Theorem* 6.2**. (correctness). Let $F(t)$ be a TC (tail-call) term for which *COMPOSE* returns the normal form $F(v)$ and the composition S. If $f = \lambda x.\ t$ and $s = \mathbf{C}[S]\, c\ s_0$ for $c \in C$ and $s_0 \in \mathit{State}$, then $s(v) = f(s_0(x))$.

Syntactic Domains

Ide	Identifiers
Opr	Operators
$Exp = Ide \mid Opr(Exp, \ldots, Exp)$	Expressions
$Com = skip \mid Ide \leftarrow Exp \mid Com; Com$	Commands

Semantic Domains

V	Values
$State = Ide \rightarrow V$	State
$C = State \rightarrow State$	Command continuations
$K = V \rightarrow C$	Expression continuations

Valuations

$\mathbf{E}$: $Exp \rightarrow K \rightarrow C$
$\mathbf{C}$: $Com \rightarrow C \rightarrow C$

Semantic Equations

$$\mathbf{E}[id]\kappa s = \kappa\, s(id)\, s$$
$$\mathbf{E}[opr(E_1, \ldots, E_n)]\kappa = \mathbf{E}[E_1]\,\lambda v_1. \cdots \mathbf{E}[E_n]\,\lambda v_n.\, \kappa(opr(v_1, \ldots, v_n))$$
$$\mathbf{C}[skip]\, c\, s = s$$
$$\mathbf{C}[id \leftarrow E]\, c = \mathbf{E}[E]\,\lambda v, s.\, c(s[v/id])$$
$$\mathbf{C}[C_1; C_2]\, c = \mathbf{C}[C_1]\,(\mathbf{C}[C_2]\, c)$$

Figure 6.2. Continuation semantics for composition in L

Proof. The proof proceeds by induction on the number of applications of *REDUCE* in the derivation of $F(v)$ from $F(t)$.

Basis. If no applications of *REDUCE* are needed to derive $F(v)$ from $F(t)$, then $t = v$, $S = skip$, and $f = \lambda v.\, v$. Therefore, $s = \mathbf{C}[skip]\, c\, s_0 = s_0$ and

$$s(v) = s_0(v) = f(s_0(v))$$

Inductive Step. Let $t = g(t')$ for a trivial operator g and a trivial term t'. Suppose *COMPOSE* reduces $g(t')$ to the normal form $g(u)$ producing the composition C_0. In the next application of *REDUCE* to $F(g(u))$, a *fetch-store* relation R is con-

structed. If R is not a partial order then *LINEAR* fails. So suppose R is a partial order and that *LINEAR* embeds it in a consistent linear order from which *COLLAPSE* constructs the composition $S' = C_1;\ v \leftarrow g\ u;\ C_2$. Then $S = C_0;\ S'$ and s becomes

$$
\begin{aligned}
&\mathbf{C}[C_0;\ S']\, c\, s_0 \\
&= \mathbf{C}[C_0]\,(\mathbf{C}[S']\, c)\, s_0 \\
&= \mathbf{C}[C_1;\ v \leftarrow g\ u;\ C_2]\, c\, r && \text{if } r = \mathbf{C}[C_0]\,(\mathbf{C}[S']\, c)\, s_0 \\
&= \mathbf{C}[C_1]\,(\mathbf{C}[v \leftarrow g\ u;\ C_2]\, c)\, r \\
&= \mathbf{C}[v \leftarrow g\ u;\ C_2]\, c\, r' && \text{if } r' = \mathbf{C}[C_1]\,(\mathbf{C}[v \leftarrow g\ u;\ C_2]\, c)\, r \\
&= \mathbf{C}[C_2]\, c\, r'[g(r'(u))/v]
\end{aligned}
$$

By *COLLAPSE*, $v \leftarrow g\ u$ does not stand in relation *fetch-store* to any assignment in C_1. Therefore, $r'(u) = r(u)$. To show $s(v) = r'(v)$, there are two cases to consider.

Case I. $(v \neq u)$. If $v \neq u$ then v is a unique variable whose only occurrence is in $v \leftarrow g\ u$. Thus, $s(v) = r'(v)$ since v does not occur in C_2.

Case II. $(v = u)$. If $s(u) \neq r'(u)$ then by *FETCH-STORE*, there is an assignment in C_2 of the form $u \leftarrow h\ u$ for some operator h, which denies that R is a partial order since $u \leftarrow g\ u < u \leftarrow h\ u$ and $u \leftarrow h\ u < u \leftarrow g\ u$. Thus, $s(u) = r'(u)$.

Therefore, $s(v) = r'(v) = g(r'(u)) = g(r(u))$. Let $k = \lambda x.t'$. The normal form $g(u)$ can be derived from $g(t')$ in fewer applications of *REDUCE* than are needed to derive $F(v)$ from $F(g(t'))$. So by the inductive hypothesis, $r(u) = k(s_0(x))$. Thus,

$$s(v) = g(r(u)) = g(k(s_0(x))) = f(s_0(x)) \quad \square$$

7. CONCRETE CODE GENERATION

L code is an intermediate representation from which concrete code can be generated. Generating concrete code involves expanding every L declaration and L assignment to concrete form. Some postprocessing of the generated code will be necessary. L declarations are expanded to an abstract syntax as are conditional, goto, and label statements. The abstract syntax must be converted to concrete syntax, a step whose complexity depends on the concrete language. For the C programming language for instance, the conversion can be done by mere macro expansion, but for Pascal, the conversion entails generating label declarations. In the spirit of keeping the instantiator language-independent, we envision a modest backend for some imperative languages to handle the conversion to concrete syntax.

Concrete code is generated from L code relative to a representation typing RT and an implementation environment Λ, the same typing and environment used in the derivation of the L code. It is the implementation environment that determines the concrete language. For example, C code is generated from the L code of Figure 6.1 since this L code was derived using a C implementation environment.

7.1. Declaration Expansion

Expanding an L declaration entails replacing the specified representation type by the concrete type it prescribes in Λ. If the specified type matches the type of a declared data representation (D_C, T_C) in Λ, then T_C is the concrete type it prescribes

in Λ. For example, consider the declaration

$$\textbf{decl}\ y_1\ \textit{repty}\,(\textit{chr}, \textit{repchr})$$

of the L code in Figure 6.1. This code was derived using the implementation environment defined by the declared implementations in Section 2 of the Appendix. Since the representation type *repty* (*chr*, *repchr*) matches the type of the declared data representation (_, `char`) in this environment, it prescribes the concrete type `char`. Therefore, the declaration expands to **decl** y_1 `char` which requires that the variable and concrete type be transposed in converting to a concrete C declaration.

7.1.1. Polymorphism

A polymorphic data representation is characterized by a concrete type scheme (Section 3.1.1). If a representation type matches the type of a declared polymorphic data representation (D_C, T_C) in Λ, then the concrete type it prescribes in Λ is an instance of the concrete type scheme T_C. An instance is created by a replacement on (D_C, T_C). Every occurrence of a representation type variable in D_C must be replaced by a concrete type. Furthermore, all occurrences of identifiers annotated with the prefix `$` must be replaced by a new name. For example, consider the declaration

$$\textbf{decl}\ x_2\ \textit{repty}\,(\textit{seq}\,(\textit{repty}\,(\textit{chr}, \textit{repchr})), \textit{useq})$$

of the L code in Figure 6.1. The specified representation type matches the type *repty* (*seq* (`*0`), *useq*) of the declared polymorphic data representation

```
(struct $us {FILE *fp; *0 buf;};, struct $us)        (7.1)
```

with `*0` bound to *repty* (*chr*, *repchr*). So the concrete type it prescribes is an instance of the concrete type scheme `struct $us`. An instance is created by

replacing the only occurrence of `*O` by `char`, the concrete type prescribed by *repty* (*chr*, *repchr*), and replacing every occurrence of `$us` by a new name, say `usOO`. With a newly-created instance `struct usOO`, the declaration expands to

```
struct usOO {FILE *fp; char buf;};
decl x2 struct usOO
```

Note that the variable declaration is now preceded by a type declaration for the newly-created instance. In practice, the instantiator actually collects all such type declarations and places them together in the output stream.

All instances of a representation type share the same instance of a concrete type scheme. So for example, if another variable were declared with the same representation type as that used in the declaration of x_2, then the instance `struct usOO` would also be used in its expanded declaration.

As *L* declarations grow more complex, so do their expansions. Consider the following declaration for a variable *x* that denotes a higher-order sequence of characters.

```
decl x repty (seq (
                repty (seq (
                      repty (chr, repchr)),
                useq)),
          useq)
```

The representation type specified in this declaration also matches the type of the declared data representation (7.1). Therefore, the declaration expands to

```
struct usOO {FILE *fp; char buf;};
struct us01 {FILE *fp; struct usOO buf;};
decl x struct us01
```

Since both instances of *seq* are represented by *useq*, *x* is declared to be a file of pointers to other files.

7.1.2. Parameterization

If a representation type matches the type of a declared parameterized data representation (D_C, T_C) in Λ then the concrete type it prescribes in Λ is T_C defined by a type declaration formed from D_C by replacing all occurrences of parameters by the values to which they are bound. For example, consider the declaration

decl x_1 *repty* (*seq* (*repty* (*chr*, *repchr*)), *bseq* (128))

of the L code in Figure 6.1. The representation type specified in this declaration matches the type *repty* (*seq* (`*0`), *bseq* (`u`)) of the declared parameterized (also polymorphic) data representation

```
(struct $bs {*0 elem[u]; int front,rear;};, struct $bs)
```

with `*0` bound to *repty* (*chr*, *repchr*), and `u` bound to '128'. So the concrete type it prescribes is an instance of the concrete type scheme `struct $bs`. In creating an instance, the parameter `u` which declares an upper bound on array size, is replaced by '128'. With a newly-created instance `struct bs00`, the declaration expands to

```
struct bs00 {char elem[128]; int front,rear;};
decl x1 struct bs00
```

7.2. Assignment Expansion

Every abstract assignment has either the form $y \leftarrow f\ t_1, \ldots, t_n$ where $t_1, \ldots, t_n$ are trivial terms, or $y \leftarrow x$, a naked form where x is a variable or a constant of primitive type. A naked assignment $y \leftarrow x$ is expanded by selecting from Λ, a declared implementation of assignment whose type matches $x :_{RT}$. If there is no such implementation then concrete code generation fails for Λ. However, if one does

exist, say $(assign\,(v,\,u),\,I)$, then the expansion of $y \leftarrow x$ is I with every occurrence of v and u in I replaced by y and x respectively.

Expanding an assignment of the non-naked variety entails expanding the trivial terms $t_1, \ldots, t_n$. By *COLLAPSE* (Section 6.1), each can be expanded to a concrete expression. A trivial term is expanded as follows. If it is a variable or a constant of primitive type then it expands to itself. Otherwise, it has the form $g\,(r_1, \ldots, r_k)$ in which case an implementation must be selected. If $(g\ u_1, \ldots, u_k, I)$ is the declared implementation selected from Λ for this instance of g, then the term expands to I with every occurrence of u_i in I replaced by the expansion of r_i. For example, since the L code of Figure 6.1 was derived using the representation typing of Figure 5.4, the instance of *hd* in the assignment $y_1 \leftarrow hd\ x_1$ of Figure 6.1, has type

$$repty\,(seq\,(repty\,(chr,\,repchr)),\,bseq\,(128)) \rightarrow repty\,(chr,\,repchr) \tag{7.2}$$

The implementation environment in which the L code of Figure 6.1 was derived, has the following declared implementation of *hd*,

```
(hd _s, _s.elem[_s.front])                    (7.3)
```

with type $repty\,(seq\,(*0),\,bseq\,(u)) \rightarrow *0$. Since the type of *hd* matches the type of this declared implementation with `*0` bound to $repty\,(chr,\,repchr)$, and `u` bound to '128', the term $hd\,(x_1)$ expands to

$$x_1\texttt{.elem[}x_1\texttt{.front]} \tag{7.4}$$

Expansion of an assignment $y \leftarrow f\ t_1, \ldots, t_n$ can proceed in one of two ways, depending on how f is implemented. If $(assign\,(v,\,f\ u_1, \ldots, u_n),\,I)$ is the declared implementation selected from Λ for this instance of f, then the pattern indicates that

I implements the assignment *directly*. In this case, the assignment expands to I with every occurrence of u_i in I replaced by the expansion of t_i, and all occurrences of v replaced by y.

If however, f is implemented by an expression then an assignment implementation is needed. If $f :_{RT} \alpha \rightarrow \beta$ for this instance of f, then a declared assignment implementation whose type matches β, is selected from Λ. As in expanding naked assignments, if one does not exist then code generation fails for Λ. However, if one does exist, say $(assign(v, u), I')$, then $y \leftarrow f\ t_1, \ldots, t_n$ expands to I' with every occurrence of u in I' replaced by the expansion of the trivial term $f(t_1, \ldots, t_n)$, and all occurrences of v replaced by y. For example, consider the assignment $y_1 \leftarrow hd\ x_1$ of Figure 6.1. The declared implementation (7.3) is selected for this instance of *hd* since its type matches the type of *hd* as given by (7.2). But (7.3) is an expression implementation of *hd*, so the code generator seeks an implementation of assignment. The type of this assignment implementation must match *repty*(*chr*, *repchr*) since this is the type β by (7.2). The environment has the declared assignment implementation

```
(assign(y,x), y = x;)                                    (7.5)
```

whose type is *repty*(*chr*, *repchr*). Therefore, it is selected to expand the assignment $y_1 \leftarrow hd\ x_1$. By replacing `x` with the expansion of $hd(x_1)$ as given by (7.4), and `y` by y_1, the assignment $y_1 \leftarrow hd\ x_1$ expands to y_1 `= x1.elem[x1.front];`.

7.2.1. Polymorphism

Any expansion produced from a polymorphic implementation must undergo additional processing. Such an expansion contains either instances of representation type

variables, or abstract assignments (Section 3.2.1). If a representation type matches the type of a polymorphic implementation then in an expansion, every occurrence of a type variable except those serving to type abstract assignments, must be replaced by a concrete type. For example, the assignment $x_2 \leftarrow$ *apndr* $x_2\ y_4$ in the L code of Figure 6.1, expands to

```
if (!fwrite(&y4,sizeof(char),1,x2.fp)) {
      errno=EIO; perror("x2"); exit(1);
      }
```

In selecting this implementation of *apndr*, the type variable `*0` in Section 2.5 of the Appendix, gets bound to *repty* (*chr*, *repchr*) which prescribes the concrete type `char`.

Following replacement of type variables, every abstract assignment must be expanded as a naked assignment. Each abstract assignment is typed by a type variable which gets bound in a matching substitution. The type to which it is bound identifies a declared assignment implementation to be used in the expansion. For example, expansion of the assignment $x_0 \leftarrow$ *cc* 1 $y_1\ x_0$ in the L code of Figure 6.1, calls for the expansion of

```
assign((*x0.ctop->uval),y1,*0)
```

an abstract assignment which comes from the implementation of *cc* in Section 2.4 of the Appendix, with `*0` bound to the representation type *repty* (*chr*, *repchr*). Since this representation type matches the type of the declared assignment implementation (7.5), the abstract assignment expands to

```
(*x0.ctop->uval) = y1;
```

It is important to note that expanding an abstract assignment can lead to yet another expansion if the selected assignment implementation itself contains abstract

assignments; see Section 2.3 of the Appendix. So a single L assignment, though it may look innocent, can produce alot of concrete code.

7.2.2. Parameterization

Like polymorphic implementations, any expansion produced from a parameterized implementation must be processed further. If a representation type matches the type of a parameterized implementation then in an expansion, every occurrence of a parameter must be replaced by the value to which it is bound. For example, the assignment $x_1 \leftarrow tl\ x_1$ in the L code of Figure 6.1, expands to

$$x_1\texttt{.front = (}x_1\texttt{.front + 1) \% 128;}$$

In selecting this implementation of *tl*, the parameter u in Section 2.6 of the Appendix, gets bound to '128'. It is important to note that parameter replacement in an expansion occurs prior to expanding any abstract assignments, since the operands of these assignments may contain parameters.

8. SOFTWARE TEMPLATES SYSTEM

The current version of the software templates system is written in C, and runs on a VAX-11/780[1] under UNIX[2] 4.2 BSD. It is composed of an instantiator `ins` and three preprocessors. Prior to instantiating a template, the types over which it is defined must be imported and implemented. Types are imported by invoking a preprocessor `import` on a file of one or more import declarations. Data type implementation specifications are preprocessed by `mkimp` which puts implementations in a form suitable for use by `ins`.

The tail-recursive transformation phase of template instantiation is a step that need only be performed once for a template. Therefore, each template specification is transformed once into tail-recursive form by a preprocessor `factor`. The tail-recursive template can then be used in a variety of different instantiations. Each preprocessor is described in this chapter followed by a description of how to use `ins`.

8.1. Importing Types

Operators of an abstract data type are imported by invoking a preprocessor `import` on a file containing one or more import declarations. The result is an import file which represents a type environment (Section 2.1.3). Every file of import declarations must begin with declarations for integer, character and token constants. Rather

[1] VAX is a trademark of Digital Equipment Corporation.

[2] UNIX is a trademark of AT&T Bell Laboratories.

than declaring all such constants, three representation type expressions for these constants are enclosed between '[' and ']' at the beginning of the file; see Section 1 of the Appendix. Import declarations must also be specified for the operators of the predefined primitive and compound types, and for the continuation data type. In the current system, declarations for these operators reside in a single file that is included with *cpp* (C preprocessor) at the beginning of every file of import declarations.

8.2. Making Implementations

Making an implementation of an abstract data type in the templates paradigm, means preprocessing it for the instantiator with `mkimp`. The preprocessor is invoked on a file containing one or more declarations of either data representations, or operator implementations. Representation types are inferred for all operator implementations, which are processed as parameterized macro definitions. The result is an implementation file of the same name except that it is suffixed by "`.i`" denoting a preprocessed implementation file. It is good practice to avoid declaring many different abstract data type implementations in a single file. Spreading different implementations across distinct files affords template users more control over building implementation environments.

8.3. Factoring Templates

The preprocessor `factor` transforms a template specification into tail-recursive form. It takes a file containing one or more function definitions, transforms them, and places the resulting tail-recursive template in a file with the same name except that it is suffixed by "`.f`" denoting a transformed template. By convention, the first function

definition appearing in a template file is the top-level function constituting the template. All other definitions are subordinate definitions and cannot be exported (occur in an instantiation directive). A top-level function must be identified for the *SEND* discrimination step described in Section 4.4.3.

8.4. Template Instantiation

A template is instantiated relative to an instantiation context and an implementation environment. Both are used to control the tailoring of a template. The context contributes to an initial typing of the template, and the environment is searched for data representations and operator implementations. Hosting a template in an application program involves specifying 1) an implementation environment, 2) an instantiation directive, and 3) an instantiation context.

8.4.1. Implementation Environment

A template user creates an implementation environment by including one or more implementation files (files suffixed by "`.i`") in an application program with the inclusion operator '`%{}%`'. Standard implementations of the primitive, compound, and continuation data types should always be included. Desired implementations of any other abstract data types over which the template being instantiated is defined, are also included. For example, the implementation environment of the Pascal program hosting the template *rev* in Figure 8.1, contains implementations of the compound, primitive, and continuation types (lines 4 and 5), and a bounded implementation of sequences (line 6). The specified implementation files (lines 4-6) come from executing `mkimp` on the files of `lib/pascal` in Section 3 of the Appendix.

8.4.2. Instantiation Directive

An instantiation directive identifies the template to be instantiated, and one or more abstract variables that will serve as input and output variables of the generated program component. For example, the instantiation directive in Figure 8.1 (line 11) instructs the instantiator to generate a component that seeks its input from the variable `x`, and binds its result to `y`. The generated component will reverse a *copy* of `x`, and assign the result to `y`. If a template user desires an incremental update as opposed to an actual copy, then this is conveyed by an instantiation directive in which the output (target) variable is also an input variable. A directive of this form does not guarantee a component that incrementally updates a value, but rather indicates to the instantiator that if such a component is derived then it is acceptable. For instance, the directive `assign(x, rev x)` merely results in the generation of a component that will reverse a copy of `x`, and assign the result back to `x`.

```
 1  program rev(input,output);
 2  type
 3  %{
 4  "lib/pascal/union.i","lib/pascal/pair.i",
 5  "lib/pascal/base.i","lib/pascal/cont.i",
 6  "lib/pascal/bseq.i"
 7  %}
 8  var
 9  repty(seq(repty(chr,repchr)),bseq(128)) x, y;
10  begin
11  assign(y, rev x)
12  end.
```

Figure 8.1. Pascal host for *rev*

8.4.3. Instantiation Context

An instantiation context is established by specifying declarations for the abstract variables of an instantiation directive. A declaration has the form

repexpr identifier-list;

where *repexpr* is a constrained representation type expression (Section 2.1.3) in that it must be free of the mapping operator '$\rightarrow$'. The type expression specified in a declaration must match the type of a data representation in the implementation environment. For example, in the Pascal host of Figure 8.1, the abstract variables `x` and `y` of the instantiation directive are declared in line 9. The representation type specified in the declaration matches the type of the data representation *bseq* in the implementation file `lib/pascal/bseq.i`.

8.4.4. Using the Instantiator

Once a template is hosted in an application program, it can be instantiated by `ins`. If `prog` is an application program that hosts a template whose tail-recursive form is given by the file `temp.f` then

```
ins prog temp.f
```

instantiates the template; `ins` copies `prog` to `stdout` with three major replacements. 1) Concrete type declarations replace the (%{}%)-include specification. 2) Concrete declarations for variables of the generated component replace declarations for abstract variables. 3) The generated component replaces the instantiation directive. For example, suppose the preprocessor `factor` produces for the template *rev*, the file `rev.f` containing the tail-recursive template Δ_{rev} of Figure 4.5. Invoking `ins` with

the host of Figure 8.1, and `rev.f` produces the Pascal program in Section 4 of the Appendix. Note that the included files of the host (lines 3-7) are supplanted by concrete type declarations for the continuation data type (lines 3-10), and the sequence data type (lines 11-14). These declarations come from the data representations *repcont* and *bseq* of `lib/pascal`. The declaration for the abstract variables `x` and `y` of the host (line 9) is replaced by five concrete variable declarations (lines 17-21). As indicated in Chapter 7, the instantiator generates an abstract syntax, here denoted by upper-case keywords (e.g. `DECL`), in the interest of remaining language-independent. The Pascal statements in lines 24 and 25 come from the implementation of *quit* in the file `lib/pascal/cont.i`, and the statements in lines 55-60 come from the implementation of *apndr* in the file `lib/pascal/bseq.i`, and from the implementation of assignment between character variables in the file `lib/pascal/base.i`.

To illustrate the reusability of templates, suppose an application calls for a C program to reverse a character string with the reversed string being written to a file. The C host in Figure 8.2 will yield such a program when the template *rev* is instantiated. The included implementation files are now taken from the C library `lib/c` in Section 2 of the Appendix. Note that the variable `y` now denotes a character sequence represented by *useq* (line 9), making it necessary to include the implementation file `lib/c/useq.i`. Invoking `ins` with this host and `rev.f` produces the C program in Section 5 of the Appendix. Incidentally, the component to which *rev* is instantiated for this host, comes from generating concrete code for the L code of Figure 6.1 with variables renamed. The concrete code for instance, in lines 36-43 comes from expanding the abstract assignment `_0` ← *cc* `1 _7 _0`.

The template *rev* can also be reused to generate programs that reverse sequences of different objects. For instance, changing the instantiation context of Figure 8.2 (lines 8 and 9) to the context established by the declarations

```
repty(seq(repty(seq(repty(chr,repchr)),bseq(16))),bseq(128)) x;
repty(seq(repty(seq(repty(chr,repchr)),bseq(16))),useq) y;
```

produces a C program that reverses sequences whose elements themselves are sequences (a higher-order sequence). As illustrated by these examples, many different implementations of *rev* can be obtained by merely changing instantiation contexts, and reinvoking `ins`.

As a final example, suppose an application demands a C program that finds the integer value associated with a character string in an array of (string, integer) pairs. Such a program is obtained by invoking `ins` with the host of Figure 8.3, and `assoc.f`, the tail-recursive form of the template *assoc*; the generated C program is

```
 1  %{
 2  "lib/c/union.i","lib/c/pair.i",
 3  "lib/c/base.i","lib/c/cont.i",
 4  "lib/c/bseq.i","lib/c/useq.i"
 5  %}
 6
 7  main () {
 8  repty(seq(repty(chr,repchr)),bseq(128)) x;
 9  repty(seq(repty(chr,repchr)),useq) y;
10
11  assign(y, rev x)
12  }
```

Figure 8.2. C host for *rev*

given in Section 6 of the Appendix. It is interesting to note that unlike the C component generated for the instantiation of *rev*, no stack is introduced for the instantiation of *assoc*, as one would expect since *assoc* is already tail-recursive.

The examples in this chapter illustrate a multitude of different dimensions along which templates and implementations of data types can be reused. It is important to remember that we have attempted to keep the host programs intelligible by demonstrating a particular kind of template usage, one where a host program's only purpose is to host a template. In general, templates can be used to generate components of more comprehensive host programs, those with much more application-specific code.

```
 1  %{
 2  "lib/c/union.i","lib/c/pair.i",
 3  "lib/c/base.i","lib/c/cont.i",
 4  "lib/c/bseq.i"
 5  %}
 6
 7  main () {
 8  repty(union(
 9        repty(int,repint),
10        repty(tok,reptok)),
11  repunion) val;
12  repty(seq(
13        repty(pair(
14              repty(tok,reptok),
15              repty(int,repint)),
16        reppair)),
17  bseq(512)) s;
18  repty(tok,reptok) key;
19
20  assign(val, assoc key s)
21  }
```

Figure 8.3. C host for *assoc*

9. CONCLUSION

The software templates methodology is one in which algorithms can be specified independently of implementations. Algorithm specifications (templates) can be understood apart from implementations, which themselves can be studied in isolation. For a specific application, a programmer selects a template, and binds its data types to appropriate implementations. The template is then instantiated to a program component tailored to the chosen implementations. Libraries of reusable templates and implementations can be developed for use across a wide variety of applications.

9.1. Correctness

Our intent here is not to describe a proof system, but rather to indicate how correctness proofs are factored in the templates methodology. Correctness of an instantiated template is established by separately verifying the correctness of both the template and the implementations of its data types. If a template is proved correct (perhaps by subgoal induction [40]) and the implementations selected for its data types have also been proved correct (as discussed in Section 3.5) then the component to which it is instantiated is correct.

The correctness of a host program, one containing an instantiation directive, is contingent upon the compatibility of implementations with the host. Implementations selected for a template's data types in an instantiation context may be partial and moreover, may rely on preconditions for their correctness. For example, a precondi-

tion for the correctness of the partial sequence implementation in Section 2.5 of the Appendix, is that the file pointer `fp` points to a file opened for reading. An implementation I of an abstract data type T is compatible with a host program if 1) every operator of T that occurs in the template specified in the instantiation directive of the host is implemented in I, and 2) the host program ensures that the preconditions guarding the integrity of I are satisfied. In related work, Gries et al. [20] have developed proof rules for proving the correctness of programs containing abstract variables and implement directives which were discussed in Section 1.2.

9.2. Information Hiding

In the templates paradigm, implementations are selected for applications based on representations of abstract values, and performance. For instance, if a numerical computation calls for extended precision then a double precision representation of real numbers might be selected. Or, a hash table implementation of a symbol table type might be selected over a binary tree for performance reasons. But, is this the extent to which template users need be familiar with implementations? Can details remain hidden, or is it necessary for users to peek inside implementations? In general, users need not have intimate knowledge of an implementation's internal details, but some knowledge of it is needed to construct interfaces for instantiated templates.

An interface must be constructed to incorporate an instantiated template in a host program. The interface binds abstract variables to abstract values which are formed by lifting concrete values. The process of lifting a concrete value v to an abstract one $abs\,(v)$, can be viewed as a form of *type casting* whereby certain preconditions are checked and representations are transformed. As a type casting operator,

the purpose of *abs* is 1) to determine if v satisfies the preconditions on which implementations of operators over *abs* (v) rely for their correctness, and 2) to transform the representation of v to the chosen representation of abstract values if it satisfies the preconditions. Currently, template users are responsible for implementing instances of *abs*, a task that may require peeking inside implementations. Consider for example, an interface for the instantiation of the template *rev* in Section 4 of the Appendix. By the host program of Figure 8.1, the abstract variable `x` must be bound to a character sequence represented by *bseq*(`128`). Casting a stream of characters to a sequence represented in this way involves checking if the stream has fewer than `128` characters (a precondition of *bseq*(`128`)), and if it does, transforming its representation by copying its characters to `x.elem` such that `x.front` points to the first character, and `x.rear` points to the last character.

Another reason template users may peek inside implementations is to make instantiated templates more efficient. Improvements in efficiency often come as a result of removing precondition checks from the implementations of operators. If a host program ensures that these preconditions are satisfied then there is no need to check them at the level of operator implementations where the checks can become costly if they are for instance, checking preconditions that remain invariant inside a loop. If an *implementor* removes checks for certain preconditions from the operator level, then these preconditions become part of those conditions that must be checked in a type cast. For example, a check for the precondition that the file pointer `fp` points to a file opened for reading has been moved out of the implementation of *tl* and into a type cast in Section 2.5 of the Appendix.

9.3. Debugging

In the templates paradigm, instantiation may not succeed for some implementation environments. An instantiation may not succeed for two reasons: 1) an implementation of either an operator or assignment may be missing in the specified implementation environment (Sections 6.1 and 7.2), or 2) a *fetch-store* relation is constructed that is not a partial order, implying that a meaning-preserving evaluation schedule cannot be produced without copying the contents of a location (Sections 6.1 and 6.2.2). Failure for the former reason suggests that either the implementor has overlooked an operation, or the operation in question cannot be implemented efficiently for the chosen representation of abstract values. An assignment operation for instance, is not implemented between sequence variables represented by files because it involves a potentially expensive file copy. If an implementation is missing for efficiency reasons then a template user's only recourse is the selection of another implementation. Failure for the latter reason indicates a vain attempt by the instantiator to deal with incremental updates by mere evaluation reordering. That is, a cyclic *fetch-store* graph has been constructed. With assignment implementations, cycles can be broken by copying concrete values to auxiliary locations. However, the current version of the instantiator does not break cycles, so a template user must select different implementations.

Another phase of debugging in this paradigm comes at the level of instantiated templates. Suppose a template user obtains an instantiation, compiles it, and upon execution encounters a runtime error. Who is responsible for introducing the error? There are three candidates: 1) the template user, 2) the template author, and 3) the

implementor. The template user may have overlooked some preconditions for selected implementations, an oversight that would indeed affect the fidelity of implementations. Errors originating in the template can be detected by separately evaluating it. A template application can be evaluated using the axioms of abstract data types as rewrite rules. If evaluation reveals an erroneous normal form, say $hd(nil)$, then the template is in error. If neither the user of the template nor its author are at fault then the error can be attributed to the implementor.

9.4. Future Work

In the current software templates system, there is no support for composing templates. At most one instantiation directive is permitted in a single host program, so users must obtain separate instantiations for templates to be composed, and interface them by implementing any necessary type casts. This is a burden that can be eliminated if implementors are allowed to identify and implement casting operations. For example, suppose a user wishes to compose two set-valued templates S_1 and S_2 with the following declaration and instantiation directives

repty(*set*(*int*), *Heap*) x, y, z;
assign(y, S_1 x)
assign(z, S_2 y)

where *Heap* is a heap representation of sets. Two separate invocations of the instantiator, one for each instantiation directive, produce a program component that realizes the composition. Since both S_1 and S_2 operate on sets represented by heaps, no type casts are necessary. But suppose S_2 can be implemented more efficiently if sets are represented as bit vectors (e.g. set union implemented by bit-wise logical or). The

abstract variables of the instantiation directive for S_2 must now be declared in terms of a bit-vector representation, say *Bit-vector*.

repty(*set* (*int*), *Heap*) x, y;
repty(*set* (*int*), *Bit-vector*) v, z;
assign(y, S_1 x)
assign(z, S_2 v)

Composing S_1 and S_2 now requires a type cast to check if the cardinality of the set y returned by S_1 exceeds the vector size (perhaps a machine's word size), and to transform the heap representation of y to the bit-vector representation of v if this size-related precondition is satisfied. In the current system, a user is responsible for implementing the type cast from the heap representation to bit vector, however, the type cast is an operation whose implementation can be specified by an implementor. We intend to extend the current system to support composition by permitting multiple instantiation directives in a single host program, and allowing implementors to identify and implement casting operations.

Syntactic changes have been proposed to simplify the specification of both templates and instantiation contexts. Templates will be specified as sequences of typed recursion equations [4, 9, 28, 44]. The template *rev* for example, might be defined by

rev (*nil*) = *nil*;
rev (*apndl* (*h*, *t*)) = *apndr* (*rev* (*t*), *h*)

Techniques currently found in the LML compiler [4] can be extended to convert template specifications of this form to our current form. An advantage to specifying templates as recursion equations is that they can be compiled by the LML compiler and executed which facilitates debugging.

Another proposed syntactic change will permit representation type expressions to be abbreviated in instantiation contexts. The names of data representations may be omitted in which case standard representations are assumed. For example, the declaration for the abstract variable `val` in Figure 8.3 would be abbreviated as

```
int + tok  val;
```

where '+' denotes union, if *repint*, *reptok* and *repunion* are standard data representations of integer, token and union values.

Many human factors in the templates methodology have not been addressed. An unsolved problem for instance, is the templates programming environment. How can template and implementation libraries be organized and presented to users? The importance of the programming environment in gaining acceptance of templates among programmers cannot be overemphasized.

REFERENCES

1. Apt, K.R., "Ten Years of Hoare's Logic: A Survey - Part I", *ACM Trans. Prog. Lang. and Systems*, vol. 3, 4 (Oct. 1981), pp. 431-483.
2. Arsac, J. and Kodratoff, Y., "Some Techniques for Recursion Removal from Recursive Functions", *ACM Trans. Prog. Lang. and Systems*, vol. 4, 2 (April 1982), pp. 295-322.
3. Atkinson, M.P. and Morrison, R., "Procedures as Persistent Data Objects", *ACM Trans. Prog. Lang. and Systems*, vol. 7, 4 (Oct. 1985), pp. 539-559.
4. Augustsson, L., "A Compiler for Lazy ML", *Proc. ACM Symp. on LISP and Functional Programming*, Austin, TX, August 1984, pp. 218-227.
5. Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Comm. ACM*, vol. 21, 8 (Aug. 1978), pp. 613-641.
6. Bird, R.S., "Notes on Recursion Elimination", *Comm. ACM*, vol. 20, 6 (June 1977), pp. 434-439.
7. Boyle, J.M. and Muralidharan, M.N., "Program Reusability Through Program Transformation", *IEEE Trans. on Software Engineering*, vol. SE-10, 5 (Sep. 1984), pp. 574-588.
8. Burstall, R.M. and Darlington, J., "A Transformation System for Developing Recursive Programs", *J. ACM*, vol. 24, 1 (Jan. 1977), pp. 44-67.
9. Burstall, R.M., MacQueen, D.B. and Sannella, D.T., "HOPE: An Experimental Applicative Language", *Proc. 1980 LISP Conference*, Stanford, CA, pp. 136-143.
10. Cheatham, T.E., "Reusability Through Program Transformations", *IEEE Trans. on Software Engineering*, vol. SE-10, 5 (Sep. 1984), pp. 589-594.
11. Claybrook, B.G., "A Specification Method for Specifying Data and Procedural Abstractions", *IEEE Trans. on Software Engineering*, vol. SE-8, 5 (Sep. 1982), pp. 449-459.
12. Dahl, O.J., "Can Program Proving be Made Practical?", Research Report 33, University of Oslo, Institute of Informatics, May 1978.
13. Darlington, J., "The Synthesis of Implementations for Abstract Data Types, A Program Transformation Tactic", in *Computer Program Synthesis Methodologies*, A. Biermann and G. Guiho (eds.), D. Reidel Publishing Co., 1983, pp. 309-334.
14. DeChampeaux, D., "About the Paterson-Wegman Linear Unification Algorithm", *J. Computer and System Sciences*, vol. 32, 1986.

15. DeMarco, T., *Private Communication*, Jan. 1984.

16. Friedman, D.P., Haynes, C.T. and Kohlbecker, E., "Programming with Continuations", in *Program Transformation and Programming Environments*, vol. F8, P. Pepper (ed.), Springer-Verlag, 1984, pp. 263-274.

17. Gannon, J., McMullin, P. and Hamlet, R., "Data-Abstraction Implementation, Specification, and Testing", *ACM Trans. Prog. Lang. and Systems*, vol. 3, 3 (July 1981), pp. 211-223.

18. Goguen J.A., Thatcher, J.W. and Wagner, E.G., "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", in *Current Trends in Programming Methodology*, vol. 4, R.T. Yeh (ed.), Prentice Hall, 1977, pp. 80-149.

19. Gordon, M., Milner, R. and Wadsworth, C., "Edinburgh LCF", *Lecture Notes in Computer Science*, vol. 78, Springer-Verlag, New York, 1979.

20. Gries, D. and Prins, J., "A New Notion of Encapsulation", *Proc. ACM SIGPLAN Symp. on Language Issues in Programming Environments*, Seattle, WA, June 1985, pp. 131-139.

21. Guttag, J.V., Horowitz, E. and Musser, D.R., "Some Extensions to Algebraic Specifications", *Proc. ACM Conf. on Language Design for Reliable Software*, Raleigh, NC, March 1977, pp. 63-67.

22. Guttag, J.V., Horowitz, E. and Musser, D.R., "The Design of Data Type Specifications", in *Current Trends in Programming Methodology*, vol. 4, R.T. Yeh (ed.), Prentice Hall, 1978, pp. 60-79.

23. Guttag, J.V., Horowitz, E. and Musser, D.R., "Abstract Data Types and Software Validation", *Comm. ACM*, vol. 21, 12 (Dec. 1978), pp. 1048-1064.

24. Guttag, J.V. and Horning, J.J., "The Algebraic Specification of Abstract Data Types", *Acta Informatica*, vol. 10, 1978, pp. 27-52.

25. Guttag, J.V., Horning, J.J. and Wing, J.M., "Larch in Five Easy Pieces", DEC SRC Technical Report, 1985.

26. Haynes, C.T., Friedman, D.P. and Wand, M., "Continuations and Coroutines", *Proc. ACM Symp. on LISP and Functional Programming*, 1984, pp. 293-298.

27. Hoare, C.A.R., "Proof of Correctness of Data Representations", *Acta Informatica*, vol. 1, 1972, pp. 271-281.

28. Hoffman, C.M. and O'Donnell, M.J., "Programming with Equations", *ACM Trans. Prog. Lang. and Systems*, vol. 4, 1 (Jan. 1982), pp. 83-112.

29. Huet, G., "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems", *J. ACM*, vol. 27, 4 (Oct. 1980), pp. 797-821.

30. Jones, T.C., "Reusability in Programming: A Survey of the State of the Art", *IEEE Trans. on Software Engineering*, vol. SE-10, 5 (Sep. 1984), pp. 544-551.

31. Kamin, S. and Archer, M., "Partial Implementations of Abstract Data Types: A Dissenting View on Errors", *Proc. Int'l Symp. on Semantics of Data Types*, Lecture Notes in Computer Science, vol. 173, 1984, pp. 317-336.

32. Kernighan, B. and Ritchie, D., *The C Programming Language*, Prentice Hall, 1978.

33. Kieburtz, R.B., "A Type Inference System for SML", Lecture Notes, Oregon Graduate Center, April 1985.

34. Kuck, D., *The Structure of Computers and Computations - Volume 1*, John Wiley & Sons, 1978, pp. 139-144.

35. Lanergan, R.G. and Poynton, B.A., "Reusable code - The application development technique of the future", *Proc. Joint SHARE/GUIDE/ IBM Appl. Develop. Symp.*, Oct. 1979, pp. 127-136.

36. Litvintchouk, S.D. and Matsumoto, A.S., "Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification", *IEEE Trans. on Software Engineering*, vol. SE-10, 5 (Sep. 1984), pp. 544-551.

37. Majster, M.E., "Abstract Data Types and Their Specification Problem", *Theoretical Computer Science*, vol. 8, 1 (Feb. 1979), pp. 89-127.

38. Majster, M.E., "Treatment of Partial Operations in the Algebraic Specification Technique", *Proc. IEEE Conf. on Specifications of Reliable Software*, Boston, MA, April 1979, pp. 190-197.

39. Milner, R., "A Theory of Type Polymorphism in Programming", *J. Computer and System Sciences*, vol. 17, 1978, pp. 348-375.

40. Morris, J.H. and Wegbreit, B., "Subgoal Induction", *Comm. ACM*, vol. 20, 4 (April 1977), pp. 209-222.

41. Moses, J., "The function of FUNCTION in LISP", *ACM SIGSAM Bulletin*, vol. 15 (July 1970), pp. 13-27.

42. Musser, D.R., " Abstract Data Type Specification in the AFFIRM System", *IEEE Trans. on Software Engineering*, vol. SE-6, 1 (Jan. 1980), pp. 24-32.

43. Neighbors, J.M., "The Draco Approach to Constructing Software from Reusable Components", *IEEE Trans. on Software Engineering*, vol. SE-10, 5 (Sep. 1984), pp. 564-574.

44. O'Donnell, M.J., "Computing in Systems Described by Equations", *Lecture Notes in Computer Science*, vol. 58, Springer-Verlag, New York, 1977.

45. O'Donnell, M.J., "A Critique of the Foundations of Hoare-Style Programming Logic", *Comm. ACM*, vol. 25, 12 (Dec. 1982), pp. 927-934.

46. Partsch, H. and Pepper, P., "A Family of Rules for Recursion Removal", *Info. Processing Letters*, vol. 5, 6 (Dec. 1976), pp. 174-177.

47. Paterson, M.S. and Wegman, M.N., "Linear Unification", *J. Computer and System Sciences*, vol. 16, 2 (1978), pp. 158-167.

48. Perlis, A.J., General Chairman's Message, *ITT Workshop on Reusability in Programming*, Newport, RI, Sep. 1983.

49. Reddy, U.S. and Jayaraman, B., "Theory of Linear Equations Applied to Program Transformation", *Proc. Int'l Joint Conf. on AI*, 1983, pp. 10-16.

50. Reynolds, J.C., "On the Relation Between Direct and Continuation Semantics", Proc. 2nd Colloq. on Automata, Language, and Programming, *Lecture Notes in Computer Science*, vol. 14, Springer-Verlag, New York, 1974, pp. 141-156.

51. Robinson, J.A., "A Machine-Oriented Logic Based on the Resolution Principle", *J. ACM*, vol. 12, 1 (Jan. 1965), pp. 23-41.

52. Rosen, B.K., "Tree-manipulating Systems and Church-Rosser Theorems", *J. ACM*, vol. 20, 1 (Jan. 1973), pp. 160-187.

53. Scherlis, W.L., "Software Development and Inferential Programming", in *Program Transformation and Programming Environments*, vol. F8, P. Pepper (ed.), Springer-Verlag, 1984, pp. 341-346.

54. Scherlis, W.L., "Abstract Data Types, Specialization, and Program Reuse (Preliminary Version)", *Proc. IFIP Int'l Workshop on Advanced Prog. Environments*, 1986, pp. 417-440.

55. Sethi, R. and Tang, A., "Constructing Call-By-Value Continuation Semantics", *J. ACM*, vol. 27, 3 (July 1980), pp. 580-597.

56. Sherman, M., "Paragon: A Language Using Type Hierarchies for the Specification, Implementation, and Selection of Abstract Data Types", *PhD Thesis*, CMU-CS-83-147, Carnegie Mellon University, 1983.

57. Strachey, C. and Wadsworth, C.P., "Continuations: A Mathematical Semantics for Handling Full Jumps", *Tech. Monog.*, PRG-11, Oxford University, 1974.

58. Strong, H.R., "Translating Recursion Equations into Flowcharts", *J. Computer and System Sciences*, vol. 5, 1971, pp. 254-285.

59. "Reference Manual for the Ada Programming Language", *Proposed Standard Document*, U.S. Dept. of Defense, 1980.

60. Volpano, D.M. and Kieburtz, R.B., "Software Templates", *Proc. 8th Int'l Conf. on Software Engineering*, London, UK, 1985, pp. 55-60.

61. Wadler, P., "Views: A Way for Elegant Definitions and Efficient Representations to Coexist", *Res. Report*, Oxford University, 1985.

62. Walker, S.A. and Strong, H.R., "Characterizations of Flowchartable Recursions", *J. Computer and System Sciences*, vol. 7, 1973, pp. 404-447.

63. Wand, M. and Friedman, D.P., "Compiling Lambda Expressions Using Continuations and Factorizations", *J. Computer Languages*, vol. 3, 1978, pp. 241-263.

64. Wand, M., "Continuation-Based Program Transformation Strategies", *J. ACM*, vol. 27, 1 (Jan. 1980), pp. 164-180.

APPENDIX

1. Import Declarations

```
[repty(int,repint), repty(chr,repchr), repty(tok,reptok)]

import pr: *0->(*1->repty(pair(*0,*1),reppair));
import fst: repty(pair(*0,*1),reppair)->*0;
import snd: repty(pair(*0,*1),reppair)->*1;
import inl: *0->repty(union(*0,*1),repunion);
import inr: *1->repty(union(*0,*1),repunion);
import isl: repty(union(*0,*1),repunion)->repty(bool,repbool);
import outl: repty(union(*0,*1),repunion)->*0;
import outr: repty(union(*0,*1),repunion)->*1;

import cc: repty(int,!1)->(*0->(repty(cont(*0),repcont)->
          repty(cont(*0),repcont)));
import c: repty(int,!1)->(repty(cont(*0),repcont)->repty(cont(*0),repcont));
import isc: repty(int,!1)->(repty(cont(*0),repcont)->repty(bool,repbool));
import tlc: repty(cont(*0),repcont)->repty(cont(*0),repcont);
import val: repty(cont(*0),repcont)->*0;
import quit: repty(cont(*0),repcont);

import not: repty(bool,repbool)->repty(bool,repbool);
import and: repty(bool,repbool)->(repty(bool,repbool)->repty(bool,repbool));
import or: repty(bool,repbool)->(repty(bool,repbool)->repty(bool,repbool));
import true: repty(bool,repbool);
import false: repty(bool,repbool);

import lt: *0->(*0->repty(bool,repbool));
import gt: *0->(*0->repty(bool,repbool));
import eq: *0->(*0->repty(bool,repbool));
import diff: repty(int,!1)->(repty(int,!1)->repty(int,!1));
import plus: repty(int,!1)->(repty(int,!1)->repty(int,!1));
import mult: repty(int,!1)->(repty(int,!1)->repty(int,!1));
import mod: repty(int,!1)->(repty(int,!1)->repty(int,!1));
import div: repty(int,!1)->(repty(int,!1)->repty(int,!1));
```

1.1. Sequence Operators

```
import apndl: *0->(repty(seq(*0),!1)->repty(seq(*0),!1));
import apndr: repty(seq(*0),!1)->(*0->repty(seq(*0),!1));
import hd: repty(seq(*0),!1)->*0;
import tl: repty(seq(*0),!1)->repty(seq(*0),!1);
import hdr: repty(seq(*0),!1)->*0;
import tlr: repty(seq(*0),!1)->repty(seq(*0),!1);
import null: repty(seq(*0),!1)->repty(bool,repbool);
import nil: repty(seq(*0),!1);
```

2. C Implementation Library -- lib/c

2.1. /base

```
repbool(): bool [|short|]
x, y: repbool;
true = [|1|]
false = [|0|]
not x = [|!x|]
and x y = [|x && y|]
or x y = [|(x || y)|]
assign(y, x) = [|y = x;|]

repchr(): chr [|char|]
x, y: repchr;
eq x y = [|x == y|]
lt x y = [|x < y|]
gt x y = [|x > y|]
assign(y, x) = [|y = x;|]

repint(): int [|int|]
x, y: repint;
diff x y = [|(x - y)|]
plus x y = [|(x + y)|]
mult x y = [|x * y|]
div x y = [|x / y|]
mod x y = [|x % y|]
eq x y = [|x == y|]
lt x y = [|x < y|]
gt x y = [|x > y|]
assign(y, x) = [|y = x;|]

reptok(): tok [|char *|]
x, y: reptok;
eq x y = [|(strcmp(x, y) == 0)|]
lt x y = [|(strcmp(x, y) < 0)|]
gt x y = [|(strcmp(x, y) > 0)|]
assign(y, x) = [|strcpy(y, x);|]
```

2.2. /union

```
repunion(): union(*0,*1)

[|struct $_du {
       short isl;
       union {*0 left; *1 right;} uval;
};|] [|struct $_du|]

_u,_v: repunion;

isl _u = [|_u.isl|]
outl _u = [|_u.uval.left|]
outr _u = [|_u.uval.right|]
assign(_u, inl _x) = [|
       _u.isl = 1;
       assign(_u.uval.left,_x,*0)
|]
assign(_u, inr _x) = [|
       _u.isl = 0;
       assign(_u.uval.right,_x,*1)
|]
assign(_u, _v) = [|
       _u.isl = _v.isl;
       if (_v.isl) {
              assign(_u.uval.left,_v.uval.left,*0)
              }
       else {
              assign(_u.uval.right,_v.uval.right,*1)
              }
|]
```

2.3. /pair

```
reppair(): pair(*0,*1)

[|struct $_cp {*0 left; *1 right;};|]
[|struct $_cp|]

_x,_y: reppair;

fst _x = [|_x.left|]
snd _x = [|_x.right|]
assign(_y, pr _l _r) = [|
       assign(_y.left,_l,*0)
       assign(_y.right,_r,*1)
|]
assign(_y, _x) = [|
       assign(_y.left,_x.left,*0)
       assign(_y.right,_x.right,*1)
|]
```

2.4. /cont

```
#define Mem(ptr,cast,sz) if(!(ptr=cast malloc(sz))){\
fprintf(stderr,"No more core\n");exit(1);}

/*
 * A C dynamic memory allocator implementation of cont(*).
 * The type variable "*" ranges over union types.
 */

repcont(): cont(*0)

[|struct _cont {
       struct _celem {
              int cc;
              *0 *uval;
              struct _celem *ptop;
       } *ctop,*cptr;
};|] [|struct _cont|]

_z: repcont;

isc _i _z = [|(_z.ctop->cc == _i)|]
val _z = [|(*_z.ctop->uval)|]
assign(_z, quit) = [|_z.ctop=0;|]
assign(_z, tlc _z) = [|
       _z.cptr = _z.ctop;
       _z.ctop = _z.ctop->ptop;
       free(_z.cptr->uval); free(_z.cptr);
|]
assign(_z, c _i _z) = [|
       Mem(_z.cptr,(struct _celem *),sizeof(struct _celem))
       _z.cptr->ptop = _z.ctop;
       _z.ctop = _z.cptr;
       _z.ctop->cc = _i;
       _z.ctop->uval = 0;
|]
assign(_z, cc _i _x _z) = [|
       Mem(_z.cptr,(struct _celem *),sizeof(struct _celem))
       _z.cptr->ptop = _z.ctop;
       _z.ctop = _z.cptr;
       _z.ctop->cc = _i;
       Mem(_z.ctop->uval,(*0 *),sizeof(*0))
       assign((*_z.ctop->uval),_x,*0)
|]
```

2.5. /useq

```
 /*
 /* A C unbounded (file) implementation of sequence.
 /* Preconditions: fp points to a file opened for reading and
 /* buf contains the next element to be read.
  */

useq(): seq(*O)

[|struct $_us {
       FILE *fp;
       *O buf;
};|] [|struct $_us|]

_s: useq;

null _s = [|(feof(_s.fp))|]
hd _s = [|_s.buf|]
assign(_s, tl _s) = [|
       fread(&_s.buf,sizeof(*O),1,_s.fp);
|]
assign(_s, nil) = [|
       if (!(_s.fp = fopen("_s","w"))) {
              fprintf(stderr,"%s: cannot open\n","_s"); exit(1);
              }
|]
assign(_s, apndr _s _x) = [|
       if (!fwrite(&_x,sizeof(*O),1,_s.fp)) {
              errno=EIO; perror("_s"); exit(1);
              }
|]
```

2.6. /bseq

```
 /*
 /* A C bounded (circular queue) implementation of sequence.
 /* Preconditions: elem has at most u-1 elements, front points
 /* to the first element, and rear points to the last.
  */

bseq(u) : seq(*0)

[|struct $_bs {
        *0 elem[u];
        int front, rear;
};|] [|struct $_bs|]

_s, _xs, _ys: bseq;

null _s = [|(_s.front == _s.rear)|]
hd _s = [|_s.elem[_s.front]|]
hdr _s = [|_s.elem[_s.rear]|]
assign(_s, tl _s) = [|
        _s.front = (_s.front + 1) % u;
|]
assign(_s, tlr _s) = [|
        _s.rear = (_s.rear + u - 1) % u;
|]
assign(_s, nil) = [|
        _s.front = _s.rear = 0;
|]
assign(_s, apndl _x _s) = [|
        if ((_s.front = (_s.front + u - 1) % u) == _s.rear)
                fprintf(stderr,"bound u exceeded\n");
        else {
                assign(_s.elem[_s.front],_x,*0)
                }
|]
assign(_s, apndr _s _x) = [|
        if ((_s.rear = (_s.rear + 1) % u) == _s.front)
                fprintf(stderr,"bound u exceeded\n");
        else {
                assign(_s.elem[_s.rear],_x,*0)
                }
|]
assign(_ys, _xs) = [|
        for(_ys.rear = _xs.front; _ys.rear != _xs.rear;) {
                assign(_ys.elem[_ys.rear],_xs.elem[_ys.rear],*0)
                _ys.rear = (_ys.rear + 1) % u;
                }
        assign(_ys.elem[_ys.rear],_xs.elem[_ys.rear],*0)
        _ys.front = _xs.front;
|]
```

3. Pascal Implementation Library -- lib/pascal

3.1. /base

```
repbool(): bool [|boolean|]
x,y: repbool;
not x = [|not x|]
and x y = [|x and y|]
or x y = [|(x or y)|]
true = [|true|]
false = [|false|]
assign(y, x) = [|y := x;|]

repchr(): chr [|char|]
x,y: repchr;
eq x y = [|x = y|]
lt x y = [|x < y|]
gt x y = [|x > y|]
assign(y, x) = [|y := x;|]

repint(): int [|integer|]
x,y: repint;
diff x y = [|(x - y)|]
plus x y = [|(x + y)|]
mult x y = [|x * y|]
mod x y = [|x mod y|]
eq x y = [|x = y|]
lt x y = [|x < y|]
gt x y = [|x > y|]
assign(y, x) = [|y := x;|]

reptok(): tok [|alfa|]
x,y: reptok;
eq x y = [|x = y|]
lt x y = [|x < y|]
gt x y = [|x > y|]
assign(y, x) = [|y := x;|]
```

3.2. /union

```
repunion(): union(*0,*1)

[|$du = record
       case isl : boolean of
       true : (left : *0);
       false : (right : *1)
       end;
|] [|$du|]

_u,_v: repunion;

isl _u = [|_u.isl|]
outl _u = [|_u.left|]
outr _u = [|_u.right|]
assign(_u, inl _x) = [|
       _u.isl := true;
       assign(_u.left,_x,*0)
|]
assign(_u, inr _x) = [|
       _u.isl := false;
       assign(_u.right,_x,*1)
|]
assign(_u, _v) = [|
       if _v.isl then begin
              assign(_u.left,_v.left,*0)
              end
       else begin
              assign(_u.right,_v.right,*1)
              end
|]
```

3.3. /pair

```
reppair(): pair(*0,*1)

[|$cp = record left: *0; right: *1 end;|]
[|$cp|]

_x,_y: reppair;

fst _x = [|_x.left|]
snd _x = [|_x.right|]
assign(_y, pr _l _r) = [|
       assign(_y.left,_l,*0)
       assign(_y.right,_r,*1)
|]
assign(_y, _x) = [|
       assign(_y.left,_x.left,*0)
       assign(_y.right,_x.right,*1)
|]
```

3.4. /cont

```
repcont(): cont(*O)

[|celem = record
       cc : integer;
       uval : *O;
       ptop : ^celem
       end;
cont = record
       ctop, cptr, freec : ^celem
       end;
|] [|cont|]

_z: repcont;

isc _i _z = [|(_z.ctop^.cc = _i)|]
val _z = [|_z.ctop^.uval|]
assign(_z, quit) = [|
       _z.ctop := nil;
       _z.freec := nil;
|]
assign(_z, tlc _z) = [|
       with _z do begin
              cptr := ctop;
              ctop := ctop^.ptop;
              cptr^.ptop := freec;
              freec := cptr
              end
|]
assign(_z, c _i _z) = [|
       with _z do begin
              if freec = nil then new(cptr)
              else begin
                     cptr := freec;
                     freec := freec^.ptop
                     end
              cptr^.ptop := ctop;
              ctop := cptr;
              ctop^.cc := _i
              end
|]
assign(_z, cc _i _x _z) = [|
       with _z do begin
              if freec = nil then new(cptr)
              else begin
                     cptr := freec;
                     freec := freec^.ptop
                     end
              cptr^.ptop := ctop;
              ctop := cptr;
              ctop^.cc := _i
              assign(ctop^.uval,_x,*O)
              end
|]
```

3.5. /useq

```
 /*
 /* A Pascal unbounded (file) implementation of sequence.
 /* Precondition: file is reset
  */

useq(): seq(*O)

[|$us = file of *O;|]
[|$us|]

_s: useq;

null _s = [|eof(_s)|]
hd _s = [|(_s^)|]
assign(_s, tl _s) = [|get(_s);|]
assign(_s, nil) = [|rewrite(_s);|]
assign(_s, apndr _s _x) = [|
       assign((_s^),_x,*O)
       put(_s);
|]
```

3.6. /bseq

```
 /*
 /* A Pascal bounded (circular queue) implementation of sequence.
 /* Preconditions: elem has at most u-1 elements, front points
 /* to the first element, and rear points to the last.
  */

bseq(u) : seq(*0)

[|$bs = record
       elem: array [1..u] of *0;
       front, rear: integer
       end;
|] [|$bs|]

_s, _xs, _ys: bseq;

null _s = [|(_s.front = _s.rear)|]
hd _s = [|_s.elem[_s.front + 1]|]
hdr _s = [|_s.elem[_s.rear + 1]|]
assign(_s, tl _s) = [|
       _s.front := (_s.front + 1) mod u;
|]
assign(_s, tlr _s) = [|
       _s.rear := (_s.rear + u - 1) mod u;
|]
assign(_s, nil) = [|
       _s.front := 0; _s.rear := 0;
|]
assign(_s, apndl _x _s) = [|
       _s.front := (_s.front + u - 1) mod u;
       if (_s.front = _s.rear) then
              write('bound u exceeded');
       else begin
              assign(_s.elem[_s.front + 1],_x,*0)
              end
|]
assign(_s, apndr _s _x) = [|
       _s.rear := (_s.rear + 1) mod u;
       if (_s.rear = _s.front) then
              write('bound u exceeded');
       else begin
              assign(_s.elem[_s.rear + 1],_x,*0)
              end
|]
assign(_ys, _xs) = [|
       _ys.rear := _xs.front;
       while _ys.rear <> _xs.rear do begin
              assign(_ys.elem[_ys.rear + 1],_xs.elem[_ys.rear + 1],*0)
              _ys.rear := (_ys.rear + 1) mod u;
              end
       assign(_ys.elem[_ys.rear + 1],_xs.elem[_ys.rear + 1],*0)
       _ys.front := _xs.front;
|]
```

4. Pascal Program -- rev

```
 1  program rev(input,output);
 2  type
 3  celem = record
 4     cc : integer;
 5     uval : char;
 6     ptop : ^celem
 7     end;
 8  cont = record
 9     ctop, cptr, freec : ^celem
10     end;
11  bs2 = record
12     elem: array [1..128] of char;
13     front, rear: integer
14     end;
15
16  var
17  DECL(_0,cont)
18  DECL(x,bs2)
19  DECL(y,bs2)
20  DECL(_7,char)
21  DECL(_9,char)
22
23  begin
24  _0.ctop := nil;
25  _0.freec := nil;
26  LABEL(revc)
27  IF (x.front = x.rear)
28  THEN
29  y.front := 0; y.rear := 0;
30  GOTO(csend)
31  ELSE
32  _7 := x.elem[x.front + 1];
33  x.front := (x.front + 1) mod 128;
34  with _0 do begin
35     if freec = nil then new(cptr)
36     else begin
37            cptr := freec;
38            freec := freec^.ptop
39            end
40     cptr^.ptop := ctop;
41     ctop := cptr;
42     ctop^.cc := 1
43     ctop^.uval := _7;
44     end
45  GOTO(revc)
46  END
47  LABEL(send1)
48  _9 := _0.ctop^.uval;
49  with _0 do begin
50     cptr := ctop;
51     ctop := ctop^.ptop;
52     cptr^.ptop := freec;
53     freec := cptr
54     end
55  y.rear := (y.rear + 1) mod 128;
56  if (y.rear = y.front) then
57     write('bound 128 exceeded');
58  else begin
59     y.elem[y.rear + 1] := _9;
60     end
61  GOTO(csend)
62  LABEL(csend)
63  IF (_0.ctop^.cc = 1)
64  THEN
65  GOTO(send1)
66  ELSE
67  GOTO(send0)
68  END
69  LABEL(send0)
70  end.
```

5. C Program -- rev

```
 1  struct _cont {
 2     struct _celem {
 3            int cc;
 4            char *uval;
 5            struct _celem *ptop;
 6     } *ctop,*cptr;
 7  };
 8  struct _bs2 {
 9     char elem[128];
10     int front, rear;
11  };
12  struct _us3 {
13     FILE *fp;
14     char buf;
15  };
16
17  main() {
18
19  DECL(_0,struct _cont)
20  DECL(x,struct _bs2)
21  DECL(y,struct _us3)
22  DECL(_7,char)
23  DECL(_9,char)
24
25  _0.ctop=0;
26  LABEL(revc)
27  IF (x.front == x.rear)
28  THEN
29  if (!(y.fp = fopen("y","w"))) {
30     fprintf(stderr,"%s: cannot open\n","y"); exit(1);
31     }
32  GOTO(csend)
33  ELSE
34  _7 = x.elem[x.front];
35  x.front = (x.front + 1) % 128;
36  if(!(_0.cptr=(struct _celem *) malloc(sizeof(struct _celem)))){
37     fprintf(stderr,"No more core\n");exit(1);}
38  _0.cptr->ptop = _0.ctop;
39  _0.ctop = _0.cptr;
40  _0.ctop->cc = 1;
41  if(!(_0.ctop->uval=(char *) malloc(sizeof(char)))){
42     fprintf(stderr,"No more core\n");exit(1);}
43  (*_0.ctop->uval) = _7;
44  GOTO(revc)
45  END
46  LABEL(send1)
47  _9 = (*_0.ctop->uval);
48  _0.cptr = _0.ctop;
49  _0.ctop = _0.ctop->ptop;
50  free(_0.cptr->uval); free(_0.cptr);
51  if (!fwrite(&_9,sizeof(char),1,y.fp)) {
52     errno=EIO; perror("y"); exit(1);
53     }
54  GOTO(csend)
55  LABEL(csend)
56  IF (_0.ctop->cc == 1)
57  THEN
58  GOTO(send1)
59  ELSE
60  GOTO(send0)
61  END
62  LABEL(send0)
63  }
```

6. C Program -- assoc

```
 1  struct _cp2 {char * left; int right;};
 2
 3  struct _bs3 {
 4     struct _cp2 elem[512];
 5     int front, rear;
 6  };
 7  struct _du4 {
 8     short isl;
 9     union {int left; char * right;} uval;
10  };
11
12  main() {
13
14  DECL(key,char *)
15  DECL(s,struct _bs3)
16  DECL(val,struct _du4)
17
18  LABEL(assoc)
19  IF (s.front == s.rear)
20  THEN
21  val.isl = 0;
22  strcpy(val.uval.right, "fail assoc");
23  GOTO(send0)
24  ELSE
25  IF (strcmp(key, s.elem[s.front].left) == 0)
26  THEN
27  val.isl = 1;
28  val.uval.left = s.elem[s.front].right;
29  GOTO(send0)
30  ELSE
31  s.front = (s.front + 1) % 512;
32  GOTO(assoc)
33  END
34  END
35  LABEL(send0)
36  }
```

BIOGRAPHICAL NOTE

Dennis Michael Volpano was born in Madison, Wisconsin on February 6, 1958. He graduated magna cum laude from the University of Wisconsin Green Bay in 1980 with a B.S. degree in Mathematics. He then entered Purdue University where he received an M.S. degree in Computer Science. He began study at the Oregon Graduate Center in the fall of 1983 after working as a software engineer for Tektronix. In 1986 he began working in the laboratories of Microelectronics and Computer Technology Corporation (MCC) in Austin, Texas as a research scientist. His research interests include functional programming, program transformation, abstract semantics, and term rewriting systems.