An Implementation of a Typing System for Counting Instances of Software Components

Haakon Vik Nilsen

January 2005

Department of Informatics
University of Bergen
PB. 7800
N-5020 BERGEN

Acknowledgments

I would like in particular to thank Prof. Marc Bezem for his enthusiastic supervision of my work on this thesis. I also thank Dr. Erik Barendsen for a valuable discussion.

Thanks to Dag Viggo Lokøen and Michael Mortensen for skillful proof-reading, and for great companionship.

Contents

1 Introduction **3**

2 Software components **5**
2.1 Properties of software components 5
2.2 Objects and components 7
2.3 Current component technologies 7
2.3.1 Java Beans 8
2.3.2 Enterprise Java Beans 8
2.3.3 Microsoft .NET 9
2.3.4 Oberon and Component Pascal 9

3 Resource management for components **11**
3.1 The problem 11
3.2 Current approaches 13
3.2.1 The Singleton pattern 14
3.2.2 “Multitons” 15

4 A review of the component language and typing system **17**
4.1 A review of the language 17
4.2 Typing system 18
4.3 Experimental parallelization support 19
4.4 Operational semantics 20
4.5 Comparison with other composition languages 20
4.5.1 Piccola 21
4.5.2 ComponentJ 21
4.5.3 Component Plans 22

5 Type inference algorithm **24**
5.1 The type inference problem 24
5.2 Type inference algorithm for the component language 24
5.2.1 Declaration reordering 28

6 Implementation 31
6.1 Introduction to the implementation . . . 31
6.2 Datatypes . . . 32
6.3 Parser . . . 33
6.4 Type inference . . . 33
6.4.1 Multiset implementation . . . 33
6.5 Other notable functions . . . 34
6.5.1 Declaration reordering . . . 34
6.5.2 Program legalization . . . 34
6.5.3 Declaration concatenation . . . 34
6.6 Efficiency of the implementation . . . 35
6.6.1 The structure of the cache . . . 36
6.6.2 Measuring cache effects . . . 37
6.7 Weaknesses of the implementation . . . 40

7 User guide 44
7.1 comp: The batch inference program . . . 44
7.1.1 Verbose mode: `--verbose` . . . 44
7.1.2 Compact mode: `--compact` . . . 45
7.1.3 Strict mode: `--strict` . . . 45
7.1.4 Disabling the type cache: `--no-cache` . . . 45
7.1.5 Call statistics . . . 45
7.2 comptop: The interactive top-level . . . 46

8 Possible variants 48
8.1 Bounds-checking algorithm . . . 48
8.2 Bottom-up algorithm . . . 51

9 Conclusions 54
9.1 Contribution . . . 54
9.2 Further work . . . 55
9.2.1 Language extensions . . . 55
9.2.2 Development of the implementation . . . 55
9.2.3 Applications . . . 56

References 57

Chapter 1

Introduction

The idea of computer software componentry has circulated since the late 1960s. The concept has been attributed to M. Douglas McIlroy [12]. He observed that, while hardware engineers could compose units from interchangeable components and subcomponents which they would pick from extensive component catalogues from multiple vendors, the emerging software industry had no equivalent. McIlroy argued that software developers "undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and [software developers] are the crofters." He argued that there should be "mass production techniques" for components in software like there were in hardware engineering, and families of interchangeable routines for any given job, where developers could pick the routine most suitable for their particular needs, and exchange them for others later on if needs changed. McIlroy went on to implement his vision as the well-known pipes and filters mechanism in Unix shells.

The promise of component-based software development has been to enable the rapid building of applications from well-tested third-party components, even to the extent that the authoring of source code reduces to a small part of the job, and *composition* becomes the primary task of the developer.

In spite of this grand vision, it is only recently that we have seen the use of (and trade in) software components becoming more common. Popular opinions on how best to develop large, robust and maintainable computer software systems shift gradually over time, and can generally be depicted as in Figure 1.1.

The shifts between these "paradigms" have primarily been motivated by increasing complexity in software as demands grow and application domains extend. Once a system is unable to cope with changes in complexity, the system becomes hard and costly to maintain, and adding new features becomes infeasible. To better handle complexity, abstraction is paramount.

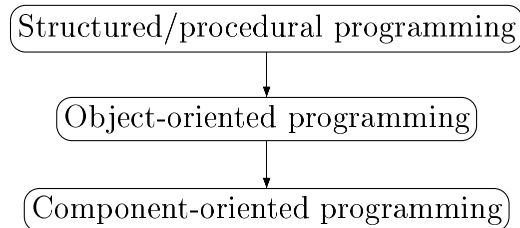


Figure 1.1: *A rudimentary depiction of shifts in software development paradigms*

The goal for a software development methodology is to create the right *kinds* of abstraction, and the appropriate *levels* of abstraction.

As software componentry becomes more prevalent, specific needs and problematic aspects become more pressing. We want not only flexibility in composition, but also robust applications which scale well.

One of the threats to robustness posed by component-based development is excessive allocation of limited resources. This can happen when inter-component dependencies make it hard to predict the resulting need for resources, and whether that need matches what is available or acceptable. [2] tackles this problem through a type system (see [5] for a general introduction to types) which counts the number of instances of components. Since this is done at development or composition time rather than at runtime, this has the potential of being useful to guide composers in designing their applications. To facilitate this, a type inference algorithm is sketched.

This thesis details our implementation of this algorithm. Chapter 2 discusses software components in general. In Chapter 3, we introduce the problem of resource management for components, and Chapter 4 provides a review of the component language introduced in [2]. In Chapter 5, we formulate the type inference algorithm. Our implementation of it is then treated in Chapter 6, while Chapter 7 gives an introduction to using the type inference tools. Chapter 8 gives some variants on the type inference algorithm to solve some problems. Finally, conclusions are given in Chapter 9.

Chapter 2

Software components

This chapter will introduce the notion of software componentry. We will present different views and common component technologies.

2.1 Properties of software components

A *software component* can be seen as a package encapsulating a limited functionality. A component exhibits a public *interface* which describes how the outside environment may interact with the component. A software component is designed to function within a particular *component framework*. The framework defines how components are expected to be implemented through interfaces or partial implementations, and may provide helpful services such as component composition, transaction management, and database connectivity to components. Components are used as software building blocks. Application developers compose applications by acquiring components "off-the-shelf" on the market and composing them into a working application, often done in graphical tools requiring little or no extra code.

Defining software components is not straight-forward. It is complicated by the fact that the term is heavily overloaded in everyday speech. Terms like "object", "module", "component", or even "component object" are often used interchangeably in reference to different notions. There is, in fact, no universally accepted definition of what software components are or what properties they need to have to qualify as components. Szyperski [8] proposes this definition of component software:

> A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Three defining characteristics are extrapolated from this: A software component

- is a *unit of independent deployment*
- is a *unit of third-party composition*
- has *no (externally) observable state* beyond what is made visible by the interface

Independent deployment refers to the assumption that a component will always be deployed (distributed) totally or not at all. Being a unit of third-party composition means that users of the component can make use of it in a larger component system without knowing, having access to knowing, or needing to know about specific implementation details of the component. Having no observable state is the final, and perhaps key characteristic. This entails that one copy of a component is indistinguishable from another and, by extension, in Szyperski's view it makes no sense to talk about "instances" of a component.

Szyperski contrasts this to our common notion of an "object", to which he assigns three characteristics analogous to those for components: an object

- is a *unit of instantiation*
- may have externally observable *state*
- *encapsulates* its state and behavior

Being units of instantiation, objects carry state, a part of which may be externally observable. This means objects have unique identities.

A more inclusive approach to the notion of software components is taken by Nierstrasz and Meijler [17]. Components, they argue, are simply software abstractions. They are in many cases higher-order entities, in that composing two components yields a third. An example is abstract classes as known from many object-oriented programming languages. These are seen as examples of higher-order components which can be partially instantiated by a second component - a subclass - to yield the third, resulting component. To form a working application, components must be *composed* and later *instantiated* - two basic operations on components recognized by [17]. Taking this approach, the authors argue one can define more flexible and conceptually natural component frameworks. This "object flavoured" view of software components is also supported by [2], and will be applied in this thesis.

2.2 Objects and components

The two approaches to software componentry we have presented may appear to conflict, although they do not. However, there are subtleties to keep in mind. Software components are collections of code (classes, functions, modules, and other definitions), and in some cases static data resources such as bitmap graphics. In the case of components consisting of classes, these classes are expected to be instantiated before they can be used, like any other class. For many such components, one of their classes acts as a point of entry to the component. When we speak of "an instance of a component", we actually mean an instance of this class. There can be many such instances, and each is an object with unique identity and state. However, the component itself needs to be loaded into the runtime environment only *once*, and has all the properties specified in [8].

Whereas components work on a compositional level, objects work on a computational level. This also implies a difference in semantics for object interfaces and component interfaces; object interfaces address computational needs, while component interfaces need to be designed to support construction of large systems. Hence, component interfaces are designed on the anticipation of working together with other components in an open and changing environment.

It is common to depict a software component as a "black box". This means that the interface specifies all services offered, but any implementation detail is hidden. The composer (ideally) does not need knowledge of such details, and in many cases does not have access to the source code, which means output can only be observed as a reaction to input. This is in contrast to objects, which are often "white box" in that implementation details are visible and depended upon, either through source code or detailed description in the documentation. An example of this is the inheritance mechanisms of object-oriented languages, where knowledge of implementation such as variable names and the order of function invocations often becomes critical. Furthermore, there have been arguments [4] for "grey-box components" where composers have some degree of implementation knowledge, to help facilitate composition and reuse. This knowledge is often encoded in special specification languages, and does not assume access to component source code.

2.3 Current component technologies

There are a number of component technologies in wide use today. We will present the most common ones here. Most of these have reached such a

"critical mass" of popularity that there are electronic market-places where trade in components happen[1].

2.3.1 Java Beans

Java Beans [7] is a software component technology for Java which provides a framework in which to run the components, or "beans". Java Beans support components with persistent state (the state can be stored and loaded into a component at runtime) and can transact between states based on messages ("events") sent to it. Components can also send such messages to other components and client objects. The component framework provides mechanisms for discovery of available components, and registration of components to make them available for such discovery.

A Java Bean in its most plain form is simply a Java class which implements the `java.io.Serializable` interface and has only the zero-argument ("default") constructor. In addition, one can define *properties* on a component. These are instance methods which follow a standardized naming convention. For example, a property "Name" of type String would be implemented as up to two methods: `public String getName()` and `public void setName(String)`. Leaving out `setName` makes the property read-only, while leaving out `getName` makes it write-only.

The component interface of a Java Bean is simply the public methods it defines. If desired, the component developer can choose to only expose a subset of those methods. Such deviation from the default behavior must be defined by providing an implementation of the `BeanInfo` interface, which can also define other meta data for the component.

Java Beans have special facilities for visual presentation and for running in a graphical user interface. On design-time, beans can be integrated in development tools to allow for easy composition by letting the user select components from a "palette", laying them out on a graphical canvas and setting property values using graphical dialog windows. On runtime, beans can be incorporated into a graphical application as a graphical component and integrates well with Java's event model for graphical user interfaces.

2.3.2 Enterprise Java Beans

Enterprise Java Beans [16], or EJB, is a server-side component architecture for the Java platform. Despite the similarity in names, it is not an extension

[1] For an example, see `http://www.componentsource.com`

of Java Beans. Whereas Java Beans is a client-side "intra-process" component model, EJB is "inter-process", and is based on distributed objects with remote method invocation (RMI) technology. EJB components run within *application servers* which implement the EJB specification. These servers offer components a standardized methods of database access, concurrency control, signalling of events to clients, data persistence, and so on.

2.3.3 Microsoft .NET

.NET [10] ("dot net") is a rich application development framework developed by Microsoft. .NET programs run on the Common Language Runtime [13], a virtual machine runtime environment which executes compiled .NET code. Since .NET offers language independence, programs can be written in any language which has a .NET compiler written for it. In addition, language integration is offered, meaning class libraries can be reused and inherited across languages.

.NET is slowly replacing Microsoft's earlier component technologies such as COM, COM+, and DCOM. These put great demands on developers, requiring them in many cases to write tedious "plumbing code" and complex registration procedures before the components could be used. In .NET, this has been simplified considerably. Components are made up by storing code and meta data in *assemblies*. These are self-contained units of deployment and versioning. The meta data describes the provided types, the inter-assembly dependencies, supported locales, and more. Versioning assures that programs which depend on a specific version of the assembly continue to work and that multiple versions of the assembly can exist side-by-side on the same system. Components can also be distributed for remote use. In this case, services are invoked with "remoting" technology, which accesses remote components through remote method invocation over a network channel such as TCP or HTTP.

2.3.4 Oberon and Component Pascal

There are also other, less prevalent component technologies, such as Oberon and Component Pascal. Oberon is an object-oriented programming language and operating environment created by Niklaus Wirth and Jürg Gutknecht at ETH Zürich in 1986, as an improvement over Modula-2 [20]. Oberon has strong support for software componentry, and because of its component-oriented design, the language and runtime environment are very small, loading more complex extensions on-demand as components. An extension of Oberon was made in 1997 and called Component Pascal, which had features

to ascertain the integrity of large component-based systems. The Black-Box Component Framework was also developed to simplify development of graphical user interface components.

Chapter 3

Resource management for components

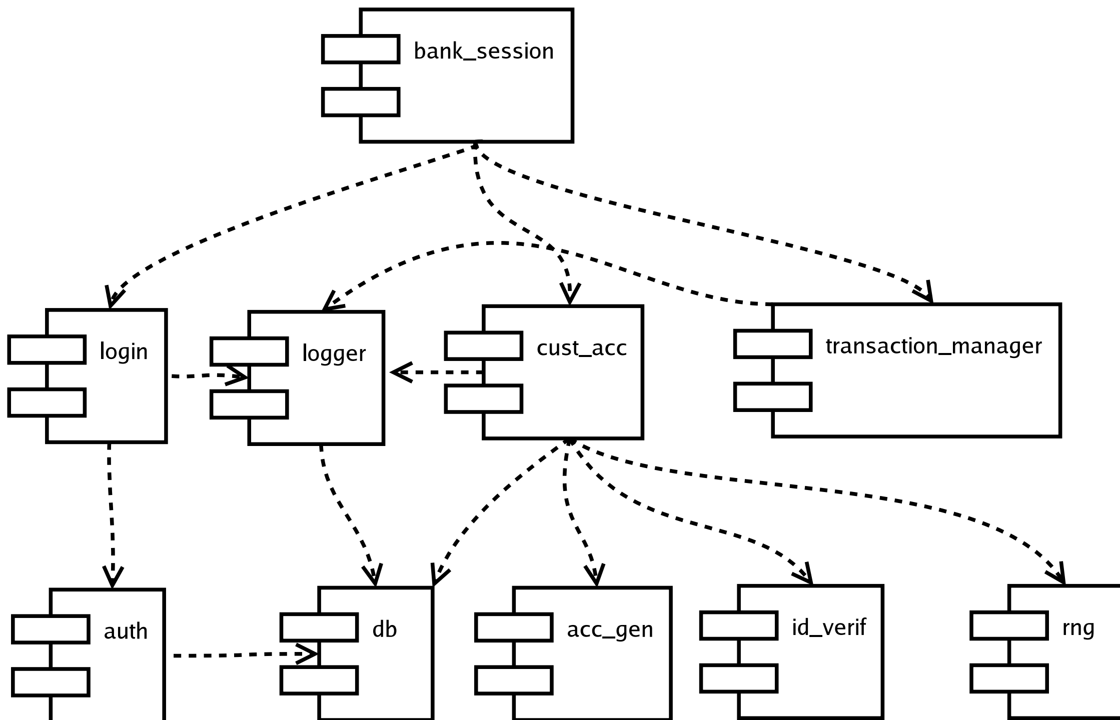
In this chapter, we introduce the problem of resource management as it pertains to software components. We give an example of the problem, and review current approaches to mend it.

3.1 The problem

Component-oriented programming lets the developer pick third-party software components off-the-shelf and assemble them in desired configurations to form a complete computer program. This approach is not only followed by the application developer at the "end of the line", but is often suitable for the developers of the components themselves. In this case, a component is composed of yet other components, and makes use of these in various patterns, various numbers of times and so on. In addition, every component which does anything meaningful, will consume certain kinds and amounts of resources during certain periods. Examples of resources are RAM memory, hard drive space and CPU time. These are scarce resources on almost any computer system. In addition, there are types of resources which are even more critical because they are based on hardware which, for mechanical or other "physical" reasons [14], can only handle a fixed amount of jobs at any given time. This could be a commodity appliance such as a printer device or a scanner, but also more specialized, business-critical devices such as a serial number generator or a cryptographically secure random number generator implemented in hardware.

This situation can put the designer of a component-based software system in a difficult position. An initial composition of the components which is

Figure 3.1: *A bank session*



conceptually sound, could have unexpected effects when later implemented, because of an over-use of scarce system resources. Ill effects could range from sub-par performance to software crashes, hardware failure, and incorrect or insecure behaviour.

Consider the component model of an Internet banking session given in Figure 3.1. While this system is overly simplified and generalized for the sake of the example, it shows a configuration of components for a system which handles an entire online banking session for a customer. Separate components handle each major function, and many of the components depend on services from one or more of the other components. To better appreciate the example, a short explanation of the reasoning behind the components is in order:

bank_session The top-level component in this system, ultimately handling control of the bank session in the various stages of operation.

db Handles database connectivity, lookups and updates.

logger Systematically logs any kind of events with timestamps to a database.

login Takes care of the logic and presentational issues behind the login procedure.

auth Authenticates the user by looking up customer data in the database. Auth consequently has a dependency on the db component.

acc_gen Generates data for establishing a new bank account in the system.

id_verif Verifies identity data with a central, authoritative registry, keyed by social security number or something equivalent.

rng a secure random number generator device which imaginably works by taking data from random noise.

cust_acc This component ties together the three aforementioned components to provide logic for establishing a new bank account on a user's request, if her identity can be verified successfully. The account information will, on successful generation, be stored in the database.

transaction_manager A quite abstract component which lets customers peruse their accounts, initiate transactions and other typical online banking tasks.

The arrows in such a component diagram define the direction of dependencies between components in such a way that the diagram makes up a dependency graph of the system. Some of the components can only stand a certain number of simultaneous instantiations. For example, `acc_gen` incorporates a serial ID generator which can have exactly one instance at a time.

A naive approach to counting the number of instances would be to count the number of in-edges to each component. However, to further complicate matters, the nature of the component dependencies may vary from actual instantiation to reuse of existing instances. Also, such a graph has no information about the chronological ordering of the instances, instance lifetime, and so on. Furthermore, the number of instances vary from run to run because of the different outcomes of user input. This makes it impossible, with no finer-grained specification tools, to tell how many instances each component will end up having.

3.2 Current approaches

There are various ways to mend the problem of resource management to a certain degree by exploiting existing features in popular development envi-

ronments.

3.2.1 The Singleton pattern

Software design patterns are textual descriptions of good solutions to recurring problems in object-oriented software design. They were introduced by Gamma, Helm, Johnson and Vlissides ("the gang of four") in [9].

One such "pattern" is the Singleton pattern, which allows programs in many object-oriented languages to control the way a class is instantiated beyond basic constructors. The pattern is most commonly used to ensure that at most (or exactly) one instantiation of a class exists at any time, and to provide a unified path of access to this object. There are many kinds of classes for which this would be appropriate; in our example with the bank session, the `db` and `logger` components could be made into "singletons" following this pattern. A sketch of a `logger` singleton implemented in Java could be:

```
public class Logger {

    private static Logger instance;

    private Logger() { }

    public static Logger getLogger() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

}
```

The most important property of this code is the *private* zero-argument constructor. In a singleton class, this should be the only constructor. While the usual way to create instances of a class is to use the `new` operator, this is not possible for a singleton class; the private constructor forbids it. Instead, the class provides a method `getLogger` which acts as the only point of access to an instance of the class. This method has, in this case, a simple conditional instantiation mechanism, and returns the single instance. Being static (belonging to the class, not instances of the class), the method provides "global" access to the singleton instance.

3.2.2 "Multitons"

A key observation regarding the Singleton pattern is that the class method providing a global, unified access to the single instance, can contain an arbitrary amount of logic to perform on each access, beyond the simple reference checking and conditional instantiation. We can exploit this to extend the pattern into allowing an arbitrary, but fixed, amount of instances. This is useful not only to limit the number of instances, but also to enforce reuse of existing instances (e.g., because the instantiation procedure needed by the class is expensive). Imagine that the `login` component is very resource demanding, and that we only have enough memory to handle at most three instances simultaneously.

```
public class Login {

  private static Login[] instances = new Login[3];
  private static boolean[] locks = new boolean[3];
  private static int count = 0;

  private int instanceNumber;

  private Login(int _instanceNumber) {
      instanceNumber = _instanceNumber;
  }

  public int getInstanceNumber() {
      return instanceNumber;
  }

  public static Login getLogin(int i) {
      if (locks[i]) {
          throw new InstanceLockedException(i);
      } else {
          locks[i] = true;
          return instances[i];
      }
  }

  public static Login newLogin() {
      Login login = null;
      if (count < instances.length) {
          instances[count] = new Login(count);
          locks[count] = true;
```

```
            login = instances[count];
            count++;
        }
        return login;
    }

    public static void free(Login l) {
        locks[l.getInstanceNumber()] = false;
    }

}
```

This extension of the Singleton pattern is sometimes informally referred to as the "Multiton pattern".

In the example, the "creational" point of entry is the `newLogin` method. This method allocates new objects and enters them into the class' instance cache, but only if the amount of previously created instances does not exceed the preset limit. Objects, once created, can be reused. However, the class attempts to ensure that each instance is used by only one client at a time by marking each instance as "locked" when it is retrieved, and the client is required to call the `free` method to unlock it when it no longer requires the instance. If this "contract" is not followed, instances will not become available for reuse. Two circumstances which have to be handled are:

- A request for a new instance is received, but the instance count is already at maximum.
- A request for an existing instance is received, but that instance has been locked.

Our example class returns the `null` reference in the first case, and raises an exception in the second.

Chapter 4

A review of the component language and typing system

This chapter provides a review of our component language, introducing its syntax and typing system. We also compare it to alternative languages and approaches built on existing research.

4.1 A review of the language

[2] introduces a small, carefully abstracted language for describing component composition. A typing system is then introduced which, through static analysis, counts the number of component instances which will be generated during run-time.

This section will review the parts which are important for this thesis.

The syntax for component programs, declarations and expressions is given by the formal definition below.

Definition (Syntax). *The syntax of the component language is defined by the following grammar. Extended Backus-Naur Form is used with the following meta-symbols: infix | for choice and underlining for Kleene closure (zero or more iterations).*

$$
\begin{array}{lcll}
Prog & ::= & Decl; Exp & \text{(Program)} \\
Decl & ::= & Var \prec Exp \; \underline{,\; Var \prec Exp} & \text{(Declarations)} \\
Exp & ::= & \epsilon & \text{(Empty Expression)} \\
 & | & \texttt{new}\, Var & \text{(New Instantiation)} \\
 & | & \texttt{reu}\, Var & \text{(Reuse Instantiation)} \\
 & | & (Exp + Exp) & \text{(Choice)} \\
 & | & \{Exp\} & \text{(Scope)} \\
 & | & Exp\ Exp & \text{(Sequential Composition)}
\end{array}
$$

A component program is built up from two parts. The declaration part is a series of assignments of component expressions to component names, each declaring a component which can be referenced in later declarations. Secondly, the *starting expression*, which is defined in terms of the definitions in the declaration part, starts execution of the program.

The expressions themselves can be built from five basic categories. `new` creates a new instance of a component. `reu` will reuse existing instances if they exist, and if they don't, an instance will be created. A *Choice* construct, e.g., $(A + B)$, represents a case where one of A and B is executed, but not both. This is used to model two phenomena. Firstly, it can model conditional execution. In this case, it mimics *if* statements as we know them from programming languages. Secondly, a choice can model nondeterminism. The *Scope* mechanism, also well-known from programming languages, introduces a block of execution in which instances created within the block are disposed of upon exit. Finally, the important *sequential composition* construct allows a compound expression to be built by juxtapositioning two expressions. We use $Var(A)$ to denote the set of components used by the expression A.

4.2 Typing system

The typing system for component programs gives the component designer a strict upper bound on the number of instances of each declared component which will be generated during run-time. *Strict* means that there exists one or more runs of the program such that the upper bound will be reached.

The set of all components $\mathbf{C}$ is partitioned into classes $\mathbf{C}_0, \ldots, \mathbf{C}_n$ such that each component in $\mathbf{C}_0$ can have an arbitrary number of active instances and each component in $\mathbf{C}_i$ with $i = 1..n$ can have at most i instances at a time. So $\mathbf{C} = \mathbf{C}_0 \cup \cdots \cup \mathbf{C}_n$ and $\mathbf{C}_i \cap \mathbf{C}_j = \emptyset$ for $0 \leq i < j \leq n$. Note that $\mathbf{C}_i$ may be empty for some i.

A *type* for a component expression is a four-tuple $\langle X^i, X^o, X^j, X^p \rangle$ of multisets over $\mathbf{C}$. Multisets assign a cardinality to each member, so that an element can appear more than once. This makes them an ideal structure for the typing system, which aims to count the highest occurring number of instances of each component. Each of the four multisets has different meaning:

X^i specifies the maximum number of instances simultaneously active at any one time during execution.

X^o specifies the "surviving" instances – those still active when execution ends. These numbers are affected by instances created within scopes.

X^j is equivalent to X^i, but under the assumption that every component has one instance already. This is to accommodate reuse in sequential composition.

X^p is equivalent to X^o, again under the assumption that every component has one instance from before.

A basis, typically denoted by Γ or Δ, is an ordered list of declarations $x_1 \prec A_1, \ldots, x_n \prec A_n$ which new components can be declared with respect to, and which component expressions can be given with respect to.

A *typing judgement* takes the form $\Gamma \vdash A : X$, meaning that the expression A has type X in the basis Γ. Legal typing judgements for component expressions are derived from the following typing rules:

Definition (Typing rules). *Typing judgments $\Gamma \vdash A : X$ are derived by the following typing rules:*

$$\textsf{Axiom}\,\frac{}{\vdash \epsilon : \langle [\,], [\,], [\,], [\,] \rangle} \qquad \textsf{Weaken}\,\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad x \notin \mathit{Dom}(\Gamma)}{\Gamma, x \prec B \vdash A : X}$$

$$\textsf{New}\,\frac{\Gamma \vdash A : X \quad x \notin \mathit{Dom}(\Gamma)}{\Gamma, x \prec A \vdash \mathtt{new}\, x : \langle X^i + x, X^o + x, X^j + x, X^p + x \rangle}$$

$$\textsf{Reu}\,\frac{\Gamma \vdash A : X \quad x \notin \mathit{Dom}(\Gamma)}{\Gamma, x \prec A \vdash \mathtt{reu}\, x : \langle X^i + x, X^o + x, X^j, X^p \rangle}$$

$$\textsf{Seq}\,\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad \forall k = 1..n. \forall c \in \mathbf{C}_k. (X^o \uplus Y^j)(c) \leq k \quad A, B \neq \epsilon}{\Gamma \vdash AB : \langle X^i \cup (X^o \uplus Y^j) \cup Y^i, (X^o \uplus Y^p) \cup Y^o, X^j \cup (X^p \uplus Y^j), X^p \uplus Y^p \rangle}$$

$$\textsf{Choice}\,\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y}{\Gamma \vdash (A + B) : X \cup Y} \qquad \textsf{Scope}\,\frac{\Gamma \vdash A : X}{\Gamma \vdash \{A\} : \langle X^i, [\,], X^j, [\,] \rangle}\ .$$

The premise for the *Seq* rule guarantees that the resulting type will not have instances exceeding the maximum allowed for each component.

4.3 Experimental parallelization support

The authors of [2] are currently working on an extension of the language to include a mechanism for running expressions in parallel. The extended syntactical category for expressions then becomes:

Definition (Extended syntax). *The extended syntax with parallelization support builds on definition 4.1 by adding to the category for expressions:*

$$\mathit{Exp} \quad ::= \quad (\mathit{Exp} || \mathit{Exp}) \quad \text{(Parallel)}$$

The typing rule for parallel composition is:

Definition (The Parallel typing rule). *Typing judgments* $\Gamma \vdash A||B : X$ *are derived by the following typing rule:*

$$\textsf{Parallel}\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad \forall k = 1..n. \forall c \in \mathbf{C}_k.(X^i \uplus Y^i)(c) \leq k}{\Gamma \vdash (A||B) : X \uplus Y}$$

In the operational semantics, $(A||B)$ means that *both* A and B will be executed, and in parallel.

It should be noted that the Seq typing rule needs further refinement to ensure that the bounds in the resulting types are strict. This is still ongoing work.

This thesis will focus primarily on the non-extended syntax, but parallelization will be brought in occasionally (and explicitly), and is also supported by the implementation (albeit with no bounds checks).

4.4 Operational semantics

[2] also defines operational semantics for the language. This is modelled as a transition system where a state is composed of a stack of multisets over component names, and a component expression. Instances are added to the multiset on the top of the stack as they are activated. When a scope expression is encountered, a new, empty multiset is pushed on top of the stack, and instances created within this scope is added to this new multiset. When the end of a scope is reached, a multiset is popped off the stack to model the deallocation of the components created within that scope.

If a program is well-typed (meaning a type can be assigned to it according to a given number of allowed instances), [2] proves that execution of the program will not create instances of any component such that it exceeds the allowed number. In other words, well-typed programs are *safe* to execute.

4.5 Comparison with other composition languages

[2] is an example of a *composition language*, .i.e., a language primarily concerned with how components are composed together, as opposed to how they are implemented. We will take rather superficial looks at other composition languages here to see what they offer with regards to resource management for component instances. This is not an exhaustive survey, but is intended to review recent research in the software components field.

4.5.1 Piccola

Piccola [1] is a composition language based on the $\pi\mathcal{L}$-calculus [11]. This calculus consists of *forms*, a kind of extensible records in which labels are bound to names, and *agents*, which perform computations and communicate forms over *channels*, asynchronously. Piccola is built on all of these notions, and combines them in a light-weight language for composition. Agents in Piccola have behaviour specified by *scripts*, and exist in a *context* which contains the legal services and forms for an agent. Piccola allows the definition of *architectural styles* which components must conform to. These define the *plugs* (exported and imported services) each component may have, and the style and rules for composition. An example architectural style provides support for the composition of Java Beans.

Piccola is weakly typed, and has no inherent notion of classes, objects, or component instances. However, because of the extensible nature of Piccola's syntax, these may all be defined on top of the existing language, as another architectural style. Some support for resource management can be envisioned as part of this, albeit not relying on a type system.

4.5.2 ComponentJ

ComponentJ[1] is a component-oriented language for modelling representation and manipulation of components. It is based on the statically typed component calculus introduced in [19]. ComponentJ has a strong emphasis on component interfaces, which components explicitly provide implementations of. Components provide services at *ports*, and component definitions include *method blocks* which can be plugged into ports with the *plug* operation. Components can be defined from scratch, or in terms of existing components by use of composition. The type system assigns static type information to components and ports, and this ensures that the consistency of the inner structure of components is retained.

Designed to work as an extension of the Java language[2], ComponentJ currently has a compiler written for it which works with Java. The compiler transforms ComponentJ definitions to Java classes and interfaces which can then be compiled by a standard Java compiler. The components can then be used with regular Java code.

ComponentJ supports the *new* keyword to create instances of components. Such statements can be embedded in method implementations within

[1]See `http://ctp.di.fct.unl.pt/~jcs/ComponentJ/`.

[2]There has been some work on a variant language for .NET, based on the same calculus. See `http://ctp.di.fct.unl.pt/~lcaires/micro/`.

the method blocks of components, or be given separately to spark off execution. Although there is no support for control over the amount of instances resulting from this in the type system, such an extension of the type system could be possible.

4.5.3 Component Plans

[3] introduces the notion of *component plans*. A component plan is provided by component authors for use by application composers, and describes typical composition patterns, or anticipated composition configurations. Component plans are specified in CoPL, which is a C-like language and is not dependent on any particular component framework or platform. These specifications are thus on an abstract level.

CoPL has components communicating through *events*. An event is "fired" by one component (the "source") and one other component can register as the event handler. As an example, a list component can fire a *Changed* event every time an element has been added to it, and this event can trigger a function call to another component, e.g. a printer component which prints information about the added element.

An important notion for component plans are *Decision Spots*. These enable interactivity by specifying abstract types or interfaces for some components and allowing the composer to select from a list of concrete implementations. The CoML generator takes care of finding the available choices. The list component in the previous example could for instance be specified as a "spot" allowing the composer to choose any available component which is type compatible with the IList interface. Alternatively, a Decision Spot can enumerate any number of interfaces, to allow both a "vertical search" (in interface inheritance hierarchies) and a "horizontal search" (across all listed types).

Application composers run CoPL specifications through a *CoML generator*, which generates CoML data. CoML is an XML[3] language which is the platform-dependent counterpart to CoPL. This means it is generated to describe components for a specific component technology, with all the technical details that demands. Being XML data means CoML can easily be processed by software tools. One intended application is "wizards" which guide the composer through the generation process by presenting possible choices for Decision Spots and generating the CoML output accordingly. CoML files can be used for documentation purposes, to exchange component compositions, or they can be transformed to code or interpreted directly (interpreters

[3] See `http://www.w3.org/XML/`.

have been written for Java and .NET).

CoPL and CoML have no inherent concept of instances in its current version. Components are perceived as collections of functions which can be called, and properties which can be set or read. It follows that bounds for instances cannot be analyzed, although with a few extensions this would be readily possible.

Chapter 5

Type inference algorithm

This chapter introduces the inference problem, particularly in relation to our component language. An algorithm for type inference is given.

5.1 The type inference problem

The type inference problem is the problem of inferring a type derivation for an expression in some language, given a set of declarations (a basis). In other words, a type inference algorithm attempts to automatically assign a type to an expression. Type inference is predominantly found in strongly-typed functional programming languages (ML dialects, Haskell). Having types inferred can relieve the programmer from having to find the type himself and annotating expressions with types to have them checked by the compiler or runtime system.

5.2 Type inference algorithm for the component language

The idea of a polynomial-time type inference algorithm for the component language was sketched in [2]. It depends heavily on the Generation Lemma presented in the paper:

Lemma 5.2.1 (Generation)

1. *If* $\Gamma \vdash \texttt{new}\, x : X$*, then* $x \in X^p$ *and there exists bases* Δ*,* Δ' *and expression* A *such that* $\Gamma = \Delta, x \prec A, \Delta'$*, and* $\Delta \vdash A : \langle X^i - x, X^o - x, X^j - x, X^p - x \rangle$*.*

2. *If* $\Gamma \vdash \texttt{reu}\, x : X$*, then* $x \in X^o$ *and there exists bases* Δ*,* Δ' *and expression* A *such that* $\Gamma = \Delta, x \prec A, \Delta'$*, and* $\Delta \vdash A : \langle X^i - x, X^o - x, X^j, X^p \rangle$*.*

3. *If* $\Gamma \vdash AB : Z$ *with* $A, B \neq \epsilon$*, then there exists* X*,* Y *such that* $\Gamma \vdash A : X$*,* $\Gamma \vdash B : Y$*,* $Z = \langle X^i \cup (X^o \uplus Y^j) \cup Y^i, (X^o \uplus Y^p) \cup Y^o, X^j \cup (X^p \uplus Y^j), X^p \uplus Y^p \rangle$*.*

4. *If* $\Gamma \vdash (A + B) : Z$*, then there exists* X*,* Y *such that* $\Gamma \vdash A : X$*,* $\Gamma \vdash B : Y$ *and* $Z = X \cup Y$*.*

5. *If* $\Gamma \vdash \{A\} : \langle X^i, [\,], X^j, [\,] \rangle$*, then there exists multisets* X^o *and* X^p *such that* $\Gamma \vdash A : \langle X^i, X^o, X^j, X^p \rangle$*.*

Sometimes referred to as an *inversion* lemma [18], a generation lemma inverts the typing rules to prove, given the value of a term, what its type must be. In other words, it allows us to make statements such as "if a term of the form f has any type, that type must be τ", and that will be apparent by following each typing rule from its conclusion to its premise. Such a lemma lends itself naturally to the construction of recursive algorithms for inferring types of compound terms, since complete types for compound terms can be computed recursively from the types of its subterms. For non-compound terms, inferring types is often trivial.

The important observation is that inferring a type for an expression E can be reduced to inferring types for $\texttt{new}\, x$ and $\texttt{reu}\, x$ for all $x \in Var(E)$. This is so because E can always be broken down into a sequence of subexpressions $E_1, \cdots, E_n$. The generation lemma lets us easily calculate the type of E if we know the types of all E_i. We also know that every E_i will be on either the form $\texttt{new}\, x$, $\texttt{reu}\, x$, $(Exp + Exp)$, $\{Exp\}$, or $(Exp || Exp)$. For $\texttt{new}\, x$ and $\texttt{reu}\, x$, the type can be found by looking up the declaration of x in the basis, inferring the type of that, and combining that with x using multiset operations as defined in the typing rules (see Section 4.2) and the generation lemma. For the other cases, E_i can be further decomposed into new levels of subexpressions, until one of the $\texttt{new}\, x$ and $\texttt{reu}\, x$ cases are reached. Thus, the problem of finding the type of an expression reduces to finding types for `new` and `reu` expressions.

Such an algorithm could easily behave exponentially if implemented naively, since duplicate instances of the same type inference problem could be generated recursively. [2] suggests storing solved instances as a solution for this.

All instances of the type inference problem can be seen as such: For $\Gamma = \Delta, x \prec E_1, \cdots, E_n, \Delta'$, inferring the type of E_i always happens with regards to the basis Δ, which is an initial segment of the basis for the original

type inference problem (the first "call" to the algorithm). Since there are polynomially many such instances, type inference can be done in polynomial time.

Based on these ideas, the algorithm INFER-TYPE has been derived. INFER-TYPE assigns types to programs of the component language, or halts with an error message if not possible.

It is important to note that INFER-TYPE assumes slightly different typing rules than those given in [2]. Specifically, the algorithm is not given upper bounds on the allowed number of instances of each component, so it knows nothing of what these bounds actually are. Instead, it computes types and reports them, so that the composer can check if the bounds are satisfactory. This implies a slight modification to the typing rules, specifically, the Seq rule:

Definition (Modified typing rules). *The Seq2 typing rule is a modification of the Seq typing rule to omit bounds checking, and is given by:*

$$\textsf{Seq2}\,\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad A, B \neq \epsilon}{\Gamma \vdash AB : \langle X^i \cup (X^o \uplus Y^j) \cup Y^i, (X^o \uplus Y^p) \cup Y^o, X^j \cup (X^p \uplus Y^j), X^p \uplus Y^p \rangle}$$

A variant of the algorithm which does bounds checking is given in Section 8.1.

Pseudo code for the algorithm is given on the next page.

INFER-TYPE(Γ, a)

1 **if** CACHE-CONTAINS(a)
2   **then return** CACHE-LOOKUP(a)
3 **if** $a = \epsilon$
4   **then** $t = \langle [\,], [\,], [\,], [\,] \rangle$
5   **else if** $a = \texttt{new}\, b$
6   **then** $x =$ LOOKUP(Γ, b)
7   **if** $x \leftarrow$ NIL
8   **then error** "Not typable"
9   **else** $t' \leftarrow$ INFER-TYPE(INITIAL-SEG$(\Gamma, b), x)$
10           $t \leftarrow \langle i(t') \uplus \{b\}, o(t') \uplus \{b\}, j(t') \uplus \{b\}, p(t') \uplus \{b\} \rangle$
11   **else if** $a = \texttt{reu}\, b$
12   **then** $x =$ LOOKUP(Γ, b)
13   **if** $x =$ NIL
14   **then error** "Not typable"
15   **else** $t' \leftarrow$ INFER-TYPE(INITIAL-SEG$(\Gamma, b), x)$
16           $t \leftarrow \langle i(t') \uplus \{b\}, o(t') \uplus \{b\}, j(t'), p(t') \rangle$
17   **else if** $a = (A + B)$
18   **then** $t_1 =$ INFER-TYPE(Γ, A)
19       $t_2 =$ INFER-TYPE(Γ, B)
20       $t = \langle i(t_1) \cup i(t_2), o(t_1) \cup o(t_2), j(t_1) \cup j(t_2), p(t_1) \cup p(t_2) \rangle$
21   **else if** $a = (A || B)$
22   **then** $t_1 \leftarrow$ INFER-TYPE(Γ, A)
23       $t_2 \leftarrow$ INFER-TYPE(Γ, B)
24       $t \leftarrow \langle i(t_1) \uplus i(t_2), o(t_1) \uplus o(t_2), j(t_1) \uplus j(t_2), p(t_1) \uplus p(t_2) \rangle$
25   **else if** $a = \{A\}$
26   **then** $t' \leftarrow$ INFER-TYPE(Γ, A)
27       $t \leftarrow \langle i(t'), \emptyset, j(t'), \emptyset \rangle$
28   **else if** $a = AB$
29   **then** $t_1 \leftarrow$ INFER-TYPE(Γ, A)
30       $t_2 \leftarrow$ INFER-TYPE(Γ, B)
31       $i' \leftarrow i(t_1) \cup (o(t_1) \uplus j(t_2))$
32       $o' \leftarrow (o(t_1) \uplus j(t_2)) \cup o(t_2)$
33       $j' \leftarrow j(t_1) \cup (p(t_1) \uplus j(t_2))$
34       $p' \leftarrow p(t_1) \uplus p(t_2)$
35       $t \leftarrow \langle i', o', j', p' \rangle$
36 CACHE-ENTER(a, t)
37 **return** t

Here, i, o, j, and p are the projection of their respective components in the type quadruple (X^i, X^o, X^j, and X^p respectively).

Lookup looks up the declaration of a component in a basis:

Definition (Lookup). *The Lookup function on bases are defined inductively by*

$$
\begin{aligned}
Lookup((x \prec A, \Gamma), x) &= A \\
Lookup(\emptyset, x) &= NIL \\
Lookup((y \prec A, \Gamma), x) &= Lookup(\Gamma, x)
\end{aligned}
$$

for a basis Γ and distinct component expressions x, y.

INITIAL-SEG computes the segment of the basis which is *initial* to some component declaration.

Definition (Basis segment initial to a component). *The basis Δ is initial to expression x_{j+1} if $\Delta = x_1 \prec A_1, \ldots, x_j \prec A_j$.*

This allows the algorithm to look up types in the "proper" initial segment instead of looking in the complete basis. While bringing the algorithm closer to the formal definitions in [2], it is not strictly a requirement for correctness, since declarations of components are distinct.

An example run of the algorithm can be illustrated by tracing the recursive steps and representing them as a call tree. For this example, we use the following simple program:

$d \prec \epsilon$,
$a \prec \texttt{new}\, d$,
$b \prec \texttt{reu}\, d(\texttt{new}\, a + \{\texttt{new}\, d\})$;
$\texttt{new}\, b$

The call tree is shown in Figure 5.1. The straight edges pointing downwards represent recursive calls, while the curved edges pointing upwards are the results. The order of the calls corresponds to a leftmost depth-first traversal of the tree. Note that due to the cache, the invocation shown in the lower right can return the type of $\texttt{new}\, d$ immediately, since this type has been computed already. When all the results have been delivered back to the root of the tree, the complete type for $\texttt{new}\, b$ can be computed (not shown in the figure).

5.2.1 Declaration reordering

The order of declarations in the component language is insignificant, however, there must exist a reordering such that the following holds:

$$\text{if } \Gamma = \Delta, x \prec A, \Delta' \text{ then } Var(A) \subseteq Dom(\Delta) \tag{5.1}$$

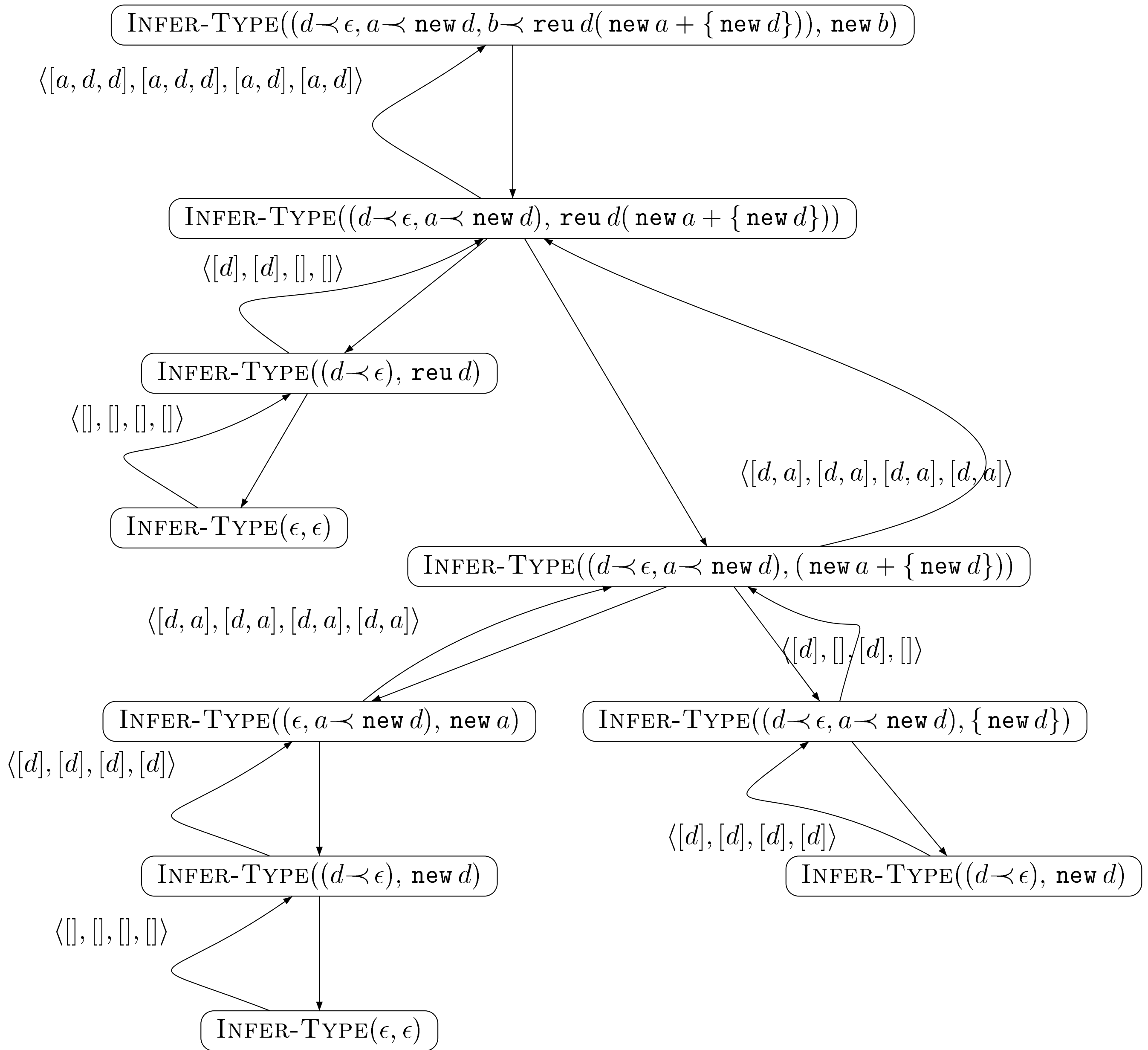


Figure 5.1: *A call tree representing an example run of the inference algorithm*

Informally, for every declaration $x \prec A$ in basis Γ, the variables occurring in A must have been declared previously. Such a reordering may not be unique (for example, in a typical opening sequence of declarations of ϵ components, the order of these declarations will not affect whether 5.1 holds).

Detecting if such a reordering exists can be done by analyzing the dependency graph of the declaration. A topological sort of the graph will yield a reordering. If no reordering can be found (i.e., the topological sort fails), that means $\Gamma = \Delta, x \prec A, \Delta'$ where either $x \prec A$ is cyclic (A depends on x, e.g. $x \prec \texttt{new}\, x$), or A depends on components which are not declared anywhere in Γ.

The type inference algorithm given depends on bases complying with (5.1). The (non-deterministic) reordering algorithm used to achieve this form is given below.

Reorder-Decl(Γ)
1 $\Delta \leftarrow$ *empty*
2 **repeat**
3 $\quad\quad$ $success \leftarrow false$
4 $\quad\quad$ **for all** $x \prec A \in \Gamma$
5 $\quad\quad\quad$ **do if** Satisfies(Δ, A)
6 $\quad\quad\quad\quad$ **then** $\Delta \leftarrow \Delta, x \prec A$
7 $\quad\quad\quad\quad\quad$ $\Gamma \leftarrow \Gamma - x \prec A$
8 $\quad\quad\quad\quad\quad$ $success \leftarrow true$
9 $\quad$ **until** $success = false$
10 **if** Empty(Γ)
11 $\quad$ **then return** Δ
12 $\quad$ **else** **error** "No possible reordering exists"

In the algorithm, Satisfies(Γ, A) checks if every component on which the expression A depends has been declared in Γ.

Chapter 6

Implementation

In this chapter, our implementation of the type inference algorithm from Chapter 5 is described on a code-level. Its efficiency is reviewed, and some weak spots are discussed.

6.1 Introduction to the implementation

The implementation of the type inference system has been written in Objective Caml[1], a popular variant of the ML programming language[2]. Such a language was found particularly fit for our purpose to implement an algorithm based on already well-defined formal specifications. Caml, as a compiled language generating native executables, has been found for many programs to be about as fast as similar programs written in C. Algebraic ("sum") datatypes and efficient recursion optimization are properties which make such a language well-suited.

The implementation includes support for parallel expressions (see Section 4.3). This was added at a late stage since it required only a small effort, although the theory currently needs further refinement to meet the requirements of strict upper bounds.

The code can be retrieved from `http://www.ii.uib.no/~haakon/components/`, and requires OCaml and GNU Make to build and run. It has been tested on Linux and Microsoft Windows.

[1] Commonly abbreviated "OCaml". See `http://caml.inria.fr/`.

[2] Historically an abbreviation for "Meta Language", a control language for theorem provers, ML was developed by Robin Milner and others in the late 1970s as a general-purpose functional programming language [15]

6.2 Datatypes

In designing the datatypes which will represent the various syntactical categories of the component language, we have tried to have them clearly reflect the structure of the language grammar.

There are three categories; `exp` representing expressions, `decl` representing component declarations, and `prog` representing complete component programs.

The definition of the `exp` type:

```
type exp =
    Empty
  | New of string * exp option
  | Reu of string * exp option
  | Choice of exp * exp * exp option
  | Parallel of exp * exp * exp option
  | Scope of exp * exp option
```

`Empty` is the type constructor representing the empty expression ϵ. The rest of the cases use the `option` type. This is a parameterized type that comes built into OCaml. Its definition is:

```
type 'a option =
    Some of 'a
  | None
```

As such, it can be used to represent types which may not have an actual value assigned to them (the `None` case), analogous to the `null` reference in languages such as Java or C#.

The `exp` type uses `option` to build lists of expressions representing component composition. To construct an `exp` value with the `New` case, one can use

```
New ("a", None)
```

This would simply represent `new` a. Building on this, to represent `new` a `reu` a, one would use

```
New ("a", Reu ("a", None))
```

The `Choice` and `Parallel` constructors take a triple by necessity. To represent (`new` a + `new` b) `reu` a, one would use

```
Choice (New ("a"), New ("b"), Reu ("a", None))
```

The scope expression `new` $anewb$ is represented by

```
Scope (New ("a"), New ("b"))
```

The declaration type is declared as such:

```
type decl = Decl of string * exp * decl option
```

Again using the `option` type, `decl` can represent a single declaration or an arbitrary-length list of them, always terminated by `None`. As an example, $a \prec \epsilon, b \prec$ `new` a is represented by

```
Decl ("a", Empty, Decl ("b", New "a", None))
```

Finally, the `prog` type is defined:

```
type prog = Prog of decl * exp
```

As with the grammar, `prog` represents a complete component program consisting of a declaration part followed by a component expression which sparks off the execution.

6.3 Parser

The parser for component programs is written with standard OCaml tools. ocamllex is used to generate the lexer, while ocamlyacc generates the parser itself. The parser adheres strictly to the grammar in [2], with the two symbols `-<` being used in place of $\prec$. For convenience, we support expressions on the form $\{A + B\}$, which will be interpreted as $\{(A + B)\}$. In addition, support for the experimental parallel composition ($||$) has been added.

The parser generates parse trees according to the syntax datatypes.

6.4 Type inference

6.4.1 Multiset implementation

As the component language relies heavily on multisets, the implementation comes with a custom `Multiset` module, since OCaml does not provide this. The `Multiset` module is based on a modification to the standard OCaml `Set` module which was originally done by Sébastien Briais[3] and fixed for errors

[3] See `http://lamp.epfl.ch/~sbriais/`.

for use with the implementation of the typing system. As with `Set`, the `Multiset` module is based on balanced binary trees. This ensures insertion and and membership checks, the focal operations of the implementation, take time logarithmic in the number of distinct elements of the multiset.

6.5 Other notable functions

6.5.1 Declaration reordering

The function for reordering a basis according to Equation 5.1 implements the algorithm from Section 5.2.1. The function, `reorder_decl` in the `Infer` module, is called as soon as a component program has been parsed, and the reordered declaration (which order may of course equal the declaration given in the program) is substituted for the given declaration.

6.5.2 Program legalization

While 5.1 has to be complied with for all declarations, the implementation has functions for "repairing" declarations on the form $\Gamma = \Delta, s \prec A, \Delta'$ where A depends on components not declared in either Δ nor Δ'. This is done by simply inserting declarations for the missing components. More precisely, if the declaration $\Gamma = \Delta, b \prec A, \Delta'$, for every component variable $a \in Var(A)$ such that $a \notin Dom(\Delta)$, Γ will be rewritten into $\Delta, a \prec \epsilon, b \prec A, \Delta'$ to form a legal basis. We refer to this process as "legalization".

The `Infer` module has functions for *expression legalization* which, given a basis and an expression, returns the additional declarations which must be appended to the basis to make the expression typable, and *declaration legalization* which operates on bases, adding the missing pieces.

6.5.3 Declaration concatenation

Declaration concatenation is used by some front-end code to extend declaration lists, and is also depended on by the declaration reordering function.

Definition (Declaration concatenation). *The concatenation of two declaration lists is represented by infix @@, and defined inductively by*

$$\begin{aligned} (x \prec A)@@\Delta &= x \prec A, \Delta \\ (x \prec A, \Gamma)@@\Delta &= x \prec A, (\Gamma @@ \Delta) \end{aligned}$$

for bases Γ *and* Δ.

These kinds of definitions are easily translated into OCaml code:

```
let rec ( @@ ) decl1 decl2 =
  match decl1 with
    Decl (s, e, None) -> Decl (s, e, Some decl2)
  | Decl (s, e, Some d) -> Decl (s, e, Some (d @@ decl2))
```

6.6 Efficiency of the implementation

A generator program has been written which yields component programs with random composition, parameterized by length of the basis and maximal size of expressions. This program has been used to generate programs of increasing sizes, with the goal of testing performance of the type inference implementation. Results from experiments with this program will be used in this section.

Two notions of "length" are adopted by the program generator, which need to be defined:

Definition (Length of expressions). *The length of a component expression is defined inductively by*

$$\begin{aligned} &Length(\texttt{new}\, x) &&= 1\\ &Length(\texttt{reu}\, x) &&= 1\\ &Length((A+B)) &&= 1\\ &Length(A) &&= 1\\ &Length(AB) &&= Length(A) + Length(B) \end{aligned}$$

for component x, and expressions A, B.

Definition (Length of declarations). *The length of a declaration is defined inductively by*

$$\begin{aligned} &Length(x \prec A) &&= 1\\ &Length(x \prec A, \Gamma) &&= 1 + Length(\Gamma) \end{aligned}$$

for component x, expression A and declaration Γ.

Storing, or caching, the results as typing problems are solved has proved very efficient and, indeed, necessary to keep the algorithm polynomial. The cache has been implemented using the `Hashtbl` data structure which is part of the standard library of OCaml. This is a hash table module with a lookup routine which runs in constant-time unless there are hash collision, in which case it switches to a balanced binary tree search on all the colliding members, and this runs in $O(\log n)$ time (n being the number of collisions for that hash value).

6.6.1 The structure of the cache

[2] suggests structuring the cache to reflect the typing judgements used to arrive that the final type. Hence, the cache would be a table with three columns. We can illustrate this by showing the cache after having inferred a type for this example:

$d \prec \epsilon$,
$a \prec$ `new` d,
$b \prec$ `new` d `new` a;
`new` b

After running the type inference algorithm on `new` b, the three-column cache has these entries:

Γ	Exp	Type (X^i shown)
	ϵ	ϵ
$d \prec \epsilon$	`new` d	$[d]$
$d \prec \epsilon, a \prec$ `new` d	`new` a	$[a, d]$
$d \prec \epsilon, a \prec$ `new` d	`new` d `new` a	$[a, d, d]$
$d \prec \epsilon, a \prec$ `new` $d, b \prec$ `new` d `new` a	`new` b	$[a, b, d, d]$

Figure 6.1: *An example three-column cache*

For brevity, we only show the X^i component of the types (for this simple program, all the type components are equal).

When there is no given upper acceptable bounds for instances of each component, and the original basis complies with 5.1, the basis component of the typing triple has no effect on the type inferred; types can be inferred with respect to the original basis. This is possible because every component name in a basis is distinct. We can use this to simplify the implementation. For a triple-column cache, initial segments have to be computed for both insertion and lookups. This operation is linear in the length of the basis, and some effort is needed to compute the minimal initial segment for composite expressions. Using anything less than the minimum initial segment can easily result in duplicate cache entries.

The two-column cache is simply a mapping from expression to types, and avoids the complexity of basis management which is not necessary in our approach. A complementary advantage is of course less space occupied in memory during inference. Reconstruction of the steps used in the type derivation is still given by the output of the type inference function when run with the `--verbose` switch.

Keeping with our example, and switching to a two-column cache which omits the basis column, the cache looks like this after inferring a type for `new` b:

Exp	Type (X^i shown)
ϵ	ϵ
`new` d	$[d]$
`new` a	$[a, d]$
`new` d `new` a	$[a, d, d]$
`new` b	$[a, b, d, d]$

Figure 6.2: *An example two-column cache*

The data in the table is equivalent to that of Figure 8.1, but all basis information has been removed. With this mapping of the cache from expressions to types, it is understood that the mapping is always with regards to the the original basis or a sufficient initial basis of it. This works in our implementation because the basis will already have been reordered before it is passed to the `infer_type` function.

6.6.2 Measuring cache effects

By looking at the amount of recursive calls and having an idea of the complexity of operations within each recursion, one can get an idea of how well the implementation scales.

First, a series of programs were generated with increasing length of the declaration. The maximal expression length was set to 4, believed to be a reasonable length for an "average" component. The behavior can be seen in Figure 6.3.

This was compared to a run with no caching of results, the parameters remaining the same (Figure 6.4). As is apparent, the amount of recursion is generally much higher. The figure shows one extreme case when around 90 components were declared. In this particular case, around 500.000 recursions were needed. This was caused by the starting expression being defined in terms of many components which had many recursive dependencies (i.e., they were high up in the dependency tree for the components in the declaration). The result was that almost all components had to be typed many times.

The same was tried with a constant declaration length of 5, and increasing maximal length of expressions (Figure 6.5).

With relatively low expression length, the amount of recursion stays low, but rises steeply with increasing length. This can be attributed to the ef-

Figure 6.3: *Increasing declaration length, cached results*

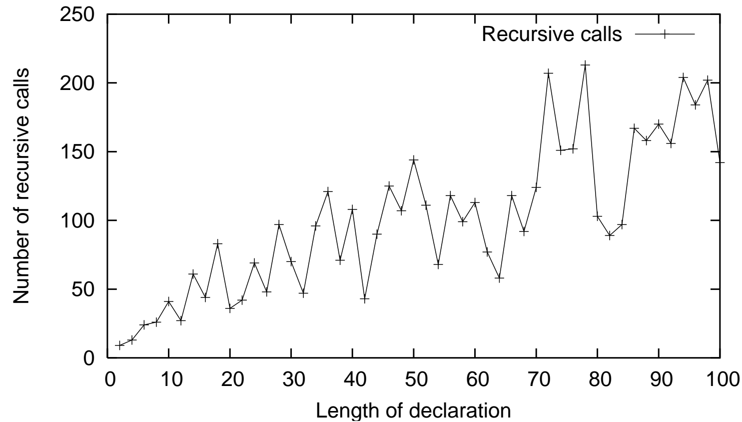
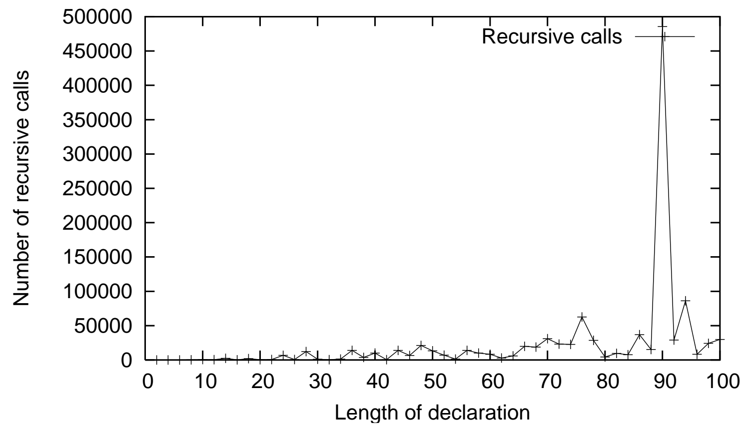


Figure 6.4: *Increasing declaration length, results not cached*

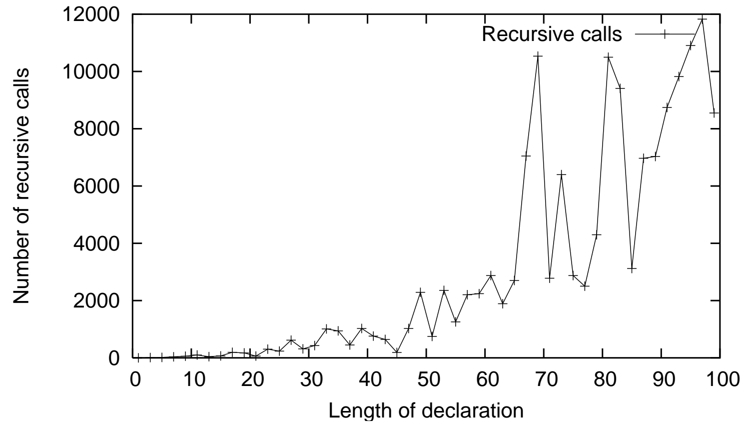


ficiency of the cache; low length of expressions leads to many similar compositions since the number of possible combinations is low. This means an expression will be more likely to have a type associated with it in the cache. If we turn off the cache and run a similar program, the situation changes radically. The recursion count often gets to the hundreds with an expression length of just 7-8, in the thousands at 11-15, and in the millions at 15 and upwards. At such lengths, the time required to infer a type is clearly exponential in the length of declarations. Some generated programs took over 30 seconds to compute a type for, while with caching, the same programs took between .5 and 1 second[4].

We can make obvious the reason for the exponential behaviour through a small example. Consider this program:

[4]Experiments run on an Intel® Pentium® 4 2.26GHz system

Figure 6.5: Increasing expression length, cached results



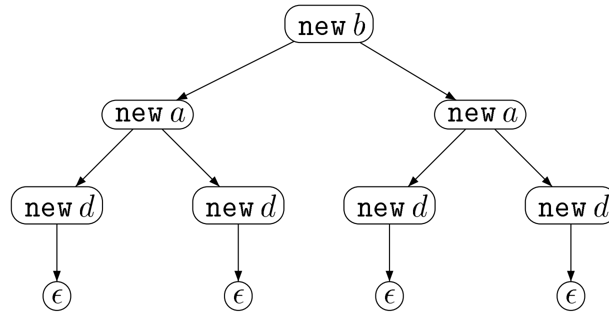


Figure 6.6: Example of type inference problems needing to be solved without cache

$d \prec \epsilon$,
$a \prec$ `new` d `new` d,
$b \prec$ `new` a `new` a;
`new` b

Without caching results, the algorithm has no memory of any of the previous solutions. The type inference problems needing to be solved is shown in Figure 6.6.

When we add the cache, the same program generates the behaviour shown in Figure 6.7. In this case, 5 of the 7 type inference problems from Figure 6.6 has been replaced by $O(1)$ lookups in the cache.

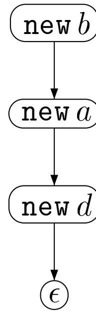


Figure 6.7: *Example of type inference problems needing to be solved with cache*

6.7 Weaknesses of the implementation

Recursive calls normally require the runtime system to keep track of branch points in the instruction code for each recursive call, so that when each call finishes, control can be returned to the proper location as frames are removed from the call stack. This requires memory space on the stack, and given enough recursion depth, which can result from large input sets, the stack will run out of space and overflow, causing, in most cases, a software crash. To fix this situation, one can apply tail recursion [6]. This involves rearranging the code such that the recursive call is the last operation in the function. A compiler can then apply tail-call optimization to transform the recursion into an iterative loop, with constant-space requirements on the stack.

The type inference algorithm relies heavily on recursive descent into datatypes which represent component expressions composed in different ways. Very long compositional sequences can cause it to overflow the stack. This limitation became apparent during experiments with component programs containing many large compositions produced by the program generator. Although composition is associative, the implementation treats it as right-associative, such that ABC is parsed as $A(BC)$. The call trees when typing the compositions thus grow towards the right. For example, the composition $ABCDEFG$ will cause the call tree shown in Figure 6.8. Each node in the tree corresponds to one stack frame required on the call stack.

Note that all components in the example composition are distinct. If there were duplicates, e.g., `new` $x \cdots$ `new` x, only the first occurrence would have been part of the call tree, as the subsequent ones would have solutions stored in the cache already. In other words, as soon as an expression is met which we have seen before, its type will be be stored in the cache, and so we can cut that branch off the call tree.

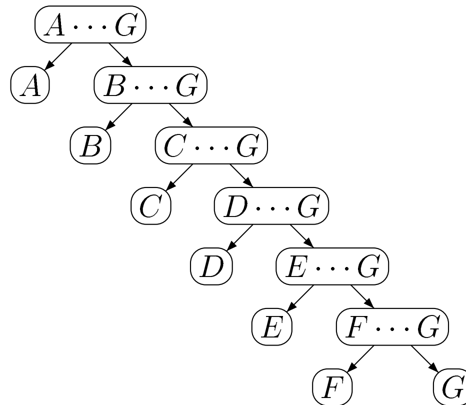


Figure 6.8: *The call tree for compositions grows linearly to the right.*

As is apparent, the need for stack space is linear in the length of expressions. Hence, very long expressions can cause stack overflows. There are measures which can be taken to improve this.

First, our algorithm recurs on all syntactic cases of expressions. This is not, however, a strict requirement. For the `new` x and `reu` x cases, recursion is necessary since we have to compute the type of the definition of x. However, for $(A + B)$ and A, we recur an extra time to arrive at the constituent expressions A and B. When such expressions are nested, this adds to the amount of stack trace needed. The implementation could be rearranged to reduce its recursive behaviour to only the `new` and `reu` cases.

Next, we show how the stack space can be reduced from linear to logarithmic in the length of expressions. This approach involves a break with the right-associative approach seen earlier. We divide sequential compositions in half and recur on each part. Returning to our example, the same composition now yields the call tree shown in Figure 6.9.

Being a full binary tree, the required stack space after this adjustment is logarithmic in the length of expressions. Since compositions are represented as sequences of expressions which have to be traversed one element at a time, dividing compositions is a linear-time operation.

Finally, the need for stack space could be eliminated completely by use of tail recursion. However, this is not easily achieved. The reason can be illustrated by a look at part of the type inference function:

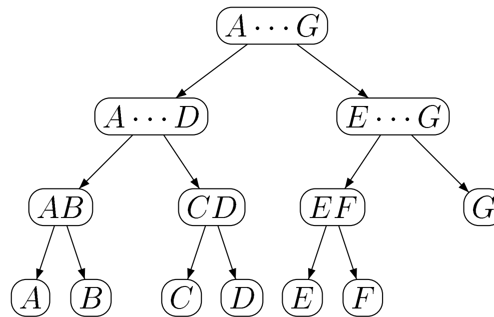


Figure 6.9: *The stack space needed can be made logarithmic.*

```
let rec infer_type =
    (* ... *)
    let inferred_type =
      match exp with
        (* ... *)
      | Choice (x1, x2, None) -> (* Generation Lemma (4) *)
          let (x1i, x1o, x1j, x1p) = infer_type env x1 in
          let (x2i, x2o, x2j, x2p) = infer_type env x2 in
          (StringSet.set_union x1i x2i, StringSet.set_union x1o x2o,
           StringSet.set_union x1j x2j, StringSet.set_union x1p x2p)
```

This characteristic sample of code shows how children of binary nodes in the parse tree (Choice and sequential composition) are handled. There is binary branching to compute the types of each of the two subexpressions, and then multiset operations to find the type of the expression as a whole (technically, of course, there are also the *let* binding operations). Because the multiset operations have to take place *after* both recursive calls, tail recursion is not possible with conventional techniques such as use of accumulators. A solution (e.g., by using continuation-passing style[5]) would likely reduce the readability of the code, which we do not consider worthwhile given that the logarithmic-size stack solution described above makes it highly unlikely the stack will ever overflow.

Using a different approach to type inference, with the BOTTOM-UP algorithm discussed in Section 8.2, tail recursion will be achieved. This does, however, incur different (although, arguably, not critical) efficiency disadvan-

[5]See `http://c2.com/cgi/wiki?ContinuationPassingStyle`.

tages.

Chapter 7

User guide

This chapter gives an introduction to the practical use of the tools developed for type inference. The type inference software mainly consists of two different programs, which will be treated separately here. Both programs make use of the same library for type inference and supporting functions.

7.1 comp: The batch inference program

`comp` is a program which reads correct and complete component programs and attempts to infer a type for the expression given in that program. For practical purposes, it can read input from both a given file and from the "standard in" stream.

To type a program stored in a file, simply run `comp filename` (with "filename" replaced by the actual file name). To pass it a program over "standard in", replace the filename argument with a dash ("-"), then type in the program or pass it a file by using the shell's redirection facilities. For most UNIX shells, the command `comp - < filename` works.

Depending on what you are interested in studying, there are various command-line options which affect the presentation of the inference process and the resulting type information:

7.1.1 Verbose mode: `--verbose`

When operating in this mode, the inference process is more "verbose"; for every recursive call as the type is tracked all the way up to the basic elements and then assembled together to form the complete type of the expression, information is printed as to what is going on. For each recurrence, a new indentation level is added as the program prints to the screen, to assist the

user in understanding the recursion strategy used.

7.1.2 Compact mode: `--compact`

The compact mode displays the resulting type in a compacted form: for each distinct member of the multiset representing the type, the cardinality is shown instead of explicitly enumerating every member. For example, instead of $[a, b, b]$, the compact mode outputs $[1 * a, 2 * b]$. This is often practical for larger programs, since the enumeration of every member of the multisets can make the type hard to read.

7.1.3 Strict mode: `--strict`

Sets strict mode for declarations. Strict mode requires that every component used in a declaration be previously defined. Unless strict mode has been set, the parser will take unknown components as "implicitly empty". This allows the designer to leave out the declarations of empty components, which may be desirable to improve readability.

7.1.4 Disabling the type cache: `--no-cache`

When set, `-no-cache` causes the type inference function to run with no caching of the type results. This is only usable for testing purposes, as it causes the function to run in exponential time.

7.1.5 Call statistics

The two options `--callstats-decl` and `--callstats-exp x` switches the type inference program from reporting inferred types to reporting call statistics for the inference function. `--callstats-exp` takes a numerical argument which is the maximal length of the expressions (to relieve the program from having to compute this). The output consists of a number representing the length of the declaration (when run with `--callstats-decl`) or the maximal expression length (when run with `--callstats-exp`), followed by the number of times the type inference function was called (including recursive calls).

The declaration list does not have to be well-ordered when given to comp as input, but such an ordering has to exist.

It is worth noting how conflicting declarations are handled. If the component x has been declared two times, once as $x \prec A$ and then as $x \prec B$, the

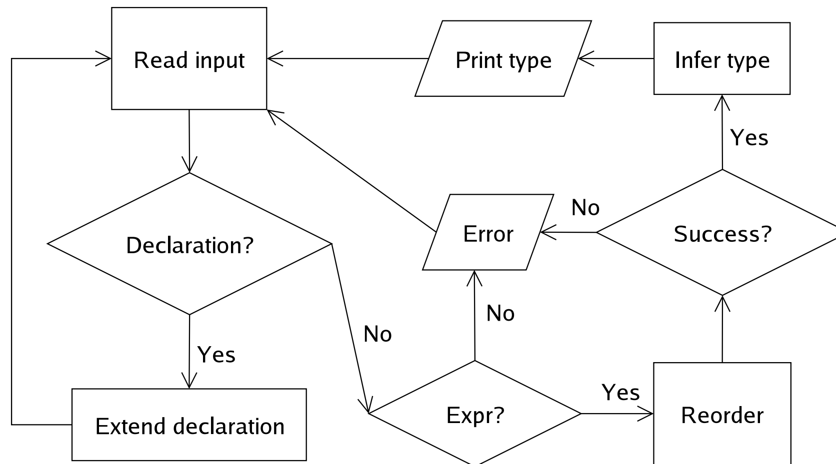


Figure 7.1: *The control flow of the comptop program*

second definition is taken as a request to handle both cases non-determinately. Thus, x is declared as $(A + B)$.

7.2 comptop: The interactive top-level

`comptop` is useful for experiments where you want to "see what happens" to the type of an expression if you change the declaration list in various ways. It is implemented as a "Read-Evaluate-Print" (REP) system, illustrated by the following flow chart:

Since `comptop` allows the user to build component programs incrementally, the syntax rules deviate slightly from the given grammar for the language. Specifically, individual declaration lines are not to be separated by commas, but are simply appended to the end of the list of declarations by entering a declaration, e.g., "b -< new a" and pressing Enter. Furthermore, there is no notion of the "last" declaration, so the declaration list is not terminated by a semicolon. To have the type of an expression inferred, simply enter that expression and a type will be inferred with regards to the current list of declarations, if possible. Having done this, one can still extend the declaration list and infer types for arbitrary expressions later on.

Let us review a sample session with comptop:

```
$ comptop
> a -< empty
a-<empty
```

```
> b -< new a
b-<new a
> c -< reu b new a
c-<reu b new a
> new c
<[a, a, b, c] | [a, a, b, c] | [a, a, c] | [a, a, c]>
>
```

Having started `comptop` and been given the “>” prompt, the user starts to enter component declarations one by one. After each, `comptop` will parse them, print back the resulting declaration as a confirmation, and enter it into the declarations list. Finally, the user enters an arbitrary component expression, which may include references to the components which have already been declared in the session. This is what triggers the type inference process, and the resulting type is immediately displayed as a response to the expression.

At any point in a `comptop` session, the user can issue the `list` command to get the current list of declarations, in the order they were introduced in by the user.

Chapter 8

Possible variants

This chapter will present some variations over the type inference algorithm.

8.1 Bounds-checking algorithm

While the implemented algorithm INFER-TYPE does not take bounds-checking into account, it can be extended to do so.

The extended bounds-checking algorithm is called INFER-TYPE-BC. The argument **C** is the class of components as defined in Section 4.2. The pseudo-code is given on the next page.

```
INFER-TYPE-BC(Γ, a, C)
 1  if CACHE-CONTAINS(a)
 2    then return CACHE-LOOKUP(a)
 3  if a = ϵ
 4    then t ← ⟨[], [], [], []⟩
 5    else if a = new b
 6           then x ← LOOKUP(INITIAL-SEG(Γ, a), b)
 7                if x ← NIL
 8                  then error "Not typable"
 9                  else t' ← INFER-TYPE-BC(Γ, x)
10                       t ← ⟨i(t') ⊎ {b}, o(t') ⊎ {b}, j(t') ⊎ {b}, p(t') ⊎ {b}⟩
11    else if a = reu b
12           then x ← LOOKUP(INITIAL-SEG(Γ, a), b)
13                if x = NIL
14                  then error "Not typable"
15                  else t' ← INFER-TYPE-BC(Γ, x)
16                       t ← ⟨i(t') ⊎ {b}, o(t') ⊎ {b}, j(t'), p(t')⟩
17    else if a = (A + B)
18           then t1 ← INFER-TYPE-BC(Γ, A)
19                t2 ← INFER-TYPE-BC(Γ, B)
20                t ← ⟨i(t1) ∪ i(t2), o(t1) ∪ o(t2), j(t1) ∪ j(t2), p(t1) ∪ p(t2)⟩
21    else if a = (A||B)
22           then t1 ← INFER-TYPE-BC(Γ, A)
23                t2 ← INFER-TYPE-BC(Γ, B)
24                t ← ⟨i(t1) ⊎ i(t2), o(t1) ⊎ o(t2), j(t1) ⊎ j(t2), p(t1) ⊎ p(t2)⟩
25    else if a = {A}
26           then t' = INFER-TYPE-BC(Γ, A)
27                t ← ⟨i(t'), ∅, j(t'), ∅⟩
28    else if a = AB
29           then t1 ← INFER-TYPE-BC(Γ, A)
30                t2 ← INFER-TYPE-BC(Γ, B)
31                for i ← 1 to length(C)
32                    do for all x ∈ C_i
33                           do if (o(t1) ⊎ j(t2))(x) > i
34                                then error "Component x exceeds boundary"
35                i' ← i(t1) ∪ (o(t1) ⊎ j(t2))
36                o' ← (o(t1) ⊎ j(t2)) ∪ o(t2)
37                j' ← j(t1) ∪ (p(t1) ⊎ j(t2))
38                p' ← p(t1) ⊎ p(t2)
39                t ← ⟨i', o', j', p'⟩
40                if notLEGAL-EXTENSION(Γ, a)
41                  then error "Illegal extension of basis"
42  CACHE-ENTER(a, t)
43  return t
```

The algorithm passes the original basis along the recursive calls. Declarations are looked up in the appropriate initial segments; if we need a type for $\texttt{new}\,x$ and $\Gamma = \Delta, x \prec A, \Delta'$, the type is looked up in Δ. When a type for

`new` x has been found, we have to check if Δ' is a legal extension of $\Delta, x \prec A$. This means checking if there is some component in Δ' which have excessive numbers of instances, enforcing the property that every component in the basis can be instantiated safely. This check is abstracted away here as a call to LEGAL-EXTENSION.

An implementation of this algorithm would require an extension to the component language syntax to include a way to specify the upper bounds for each component. We have revised the syntax to include support for this. In this revised syntax, upper bounds are given as "arguments" to each component declaration. While the bound is required, 0 can be given to have the component included in $\mathbf{C}_0$ (supporting an unbounded number of instances). A small example program in the extended syntax is:

$d(0) \prec \epsilon$,
$a(2) \prec$ `new` d,
$b(3) \prec$ `new` a `new` d;
`new` b

The bounds checking in the algorithm happens in the final case, which deals with sequential composition. The conditions in the original Seq typing rule has been used to check that when two components are composed, the resulting expression has no component exceeding its allowed number of instances. The reasoning behind this is that the Seq rule is the only typing rule which can yield expressions with increased number of instances compared to those in the premise. We have to check all $C_{1..n}$ to see if the resulting expression will have more than i instances of any component $x \in C_i$. To find the strict upper bound on the number of instances of component x a sequential composition AB can attain during a run of the composed expression, it will not suffice to look at either A nor B in isolation. The number of instances depends on the number alive *after* a run of A ($o(t_1)$), and then *during* the following run of B. We also have to consider that instances of x created by A will be reused by any `reu` expressions during the run of B, and not increase the instance count. We know that A and B both have all instances within their upper bounds since types have been derived for them. This means that at no point during execution of A will x have more than i instances, and $o(t_1)(x) \leq i$. The additional number of instances of x which can be created during a run of AB is $j(t_2)(x)$, because we can assume that x already has a surviving instance from A which carries over to B. If this is *not* the case, we will assume no instances when there is actually one, but one instance cannot exceed any legal upper bound, since there is no class of components which cannot have any instances. Hence, the composed type will never have

instance counts exceeding boundaries, and we can raise an error in our algorithm if $(o(t_1) \uplus j(t_2))(x) > i$ (or equivalently, $o(t_1)(x) + j(t_2)(x) > i$).

8.2 Bottom-up algorithm

The algorithms we have seen so far work recursively "backwards" by looking at the starting expression, determining which expressions need to be typed in order to construct the complete type, and then recurring right-to-left in the ordered basis. An alternative approach is to start the computation of types from the other end of the ordered basis – from the start – all the way through the basis, and finally computing the type of the starting expression. This has the advantage of being a conceptually simpler approach. The disadvantage of this approach is that it is less efficient in most cases, since there is the danger of computing types unnecessarily. This happens in the cases when a component is declared but never used. It can also happen because we have to compute types for both `new` x and `reu` x for every component x in the basis, at a point when we cannot know if either of them are actually needed.

The algorithm, Infer-Type-BU ("BU" abbreviates "Bottom-up"), follows.

```
Infer-Type-BU(Γ, x, C)
 1  if Γ = x≺A, Δ
 2     then Cache-Enter(A, Compute-Type(A))
 3          Cache-Enter(new x, Compute-Type(new x))
 4          Cache-Enter(reu x, Compute-Type(reu x))
 5          Infer-Type-BU(Δ, x, C)
 6     else if Γ = x≺A
 7             then Cache-Enter(A, Compute-Type(A))
 8                  Cache-Enter(new x, Compute-Type(new x))
 9                  Cache-Enter(reu x, Compute-Type(reu x))
10                  t ← Compute-Type(x)
11                  if Bounds-Ok(t, C)
12                     then return t
13                     else error "Some components has instance counts out of bounds"
```

This algorithm has the actual type inference abstracted away in the calls to the Compute-Type function, because the process is essentially the same as in the other algorithms. The only difference here is that all the results needed are already stored in the cache, so Compute-Type does not need to recur. This has the added advantage of making Infer-Type-BU tail

recursive, which means it can be expressed as an imperative loop. Thus, the problems with stack overflow for exceedingly large expressions (see Section 6.7) are avoided.

For an understanding of the functioning of INFER-TYPE-BU, we look at how the cache is built up step-by-step, each step representing an invocation of the algorithm, when running the algorithm on the following example program:

$d \prec \epsilon$,
$a \prec$ `new` d,
$b \prec ($ `new` $a +$ `reu` $d)$;
`new` b

Following the algorithm, the type of ϵ is first computed and entered into the cache, then the types of `new` d and `reu` d, and so on for the next declaration.

Step	*Exp*	Type
1	ϵ	$\langle [\,], [\,], [\,], [\,] \rangle$
1	`new` d	$\langle [d], [d], [d], [d] \rangle$
1	`reu` d	$\langle [d], [d], [\,], [\,] \rangle$
2	`new` a	$\langle [a,d], [a,d], [a,d], [a,d] \rangle$
2	`reu` a	$\langle [a,d], [a,d], [d], [d] \rangle$
3	$($ `new` $a +$ `reu` $d)$	$\langle [a,d], [a,d], [a,d], [a,d] \rangle$
3	`new` b	$\langle [a,b,d], [a,b,d], [a,b,d], [a,b,d] \rangle$
3	`reu` b	$\langle [a,b,d], [a,b,d], [a,d], [a,d] \rangle$

Figure 8.1: *The cache construction during a run of the* BOTTOM-UP *algorithm*

Bounds-checking with this algorithm has to be done carefully. We use the Seq2 typing rule for typing sequential composition (recall that this variant of the Seq rule does not check bounds). Finally, we check that the type of the starting expression is within upper bounds, which is abstracted away here as a call to the BOUNDS-OK function. The checking is done simply by counting the instances of each component in the X^i part of the type and comparing the total amount with the allowed numbers in $\mathbf{C}$. In other words, for every component $x \in X^i$, the program is well-typed if $x \in \mathbf{C}_0 \cup \ldots \cup \mathbf{C}_{X^i(x)}$.

Alternatively, the algorithm can be made to be more strict by enforcing the property that all components have to be well-typed, regardless of whether they play a role in the starting expression. More formally, for every $x \prec A \in \Gamma$, `new` x has to be well-typed, which means that we do not allow A to have

instance counts exceeding boundaries. Although such strictness may seem unnecessary, disregarding the untypability of the unused parts of a basis may cause trouble in the later stages of the design.

Chapter 9

Conclusions

9.1 Contribution

This thesis has detailed our implementation of a typing system which counts the number of software component instances which results from various kinds of composition. A simple component language tailored for this specific task was introduced in [2], and reviewed in this thesis, along with a type system for it. At the heart of our implementation is the type inference algorithm which we formulated. Running in polynomial time, this algorithm assigns a type to a program. The type consists of the number of instances each declared component can attain in different situations. Non-determinism can cause the actual number of instances to be lower, but there always exists a run of the program such that the number of instances counted in the type is attained. Running the program will never lead to a situation where the number of instances in each of the situations *exceeds* the number given in the type. Thus, the types for expressions in our component language has *strict upper bounds* on the number of instances of each component.

The implementation of the type inference algorithm has been used to create tools which are useful both for analysis of the algorithm itself, and as an application of the theory to help guide the design of component systems.

Furthermore, we have investigated two variations on the algorithm. An important one was the bounds-checking algorithm, which relies on an extension of the component language to include upper bounds on the number of instances of each component. Theory from [2] was applied to enforce these bounds. The advantages of this algorithm is that it does not rely on the composer to check that the bounds are acceptable after the typing process has finished; it is able to fail immediately if bounds are exceeded, and inform the composer of the error. We also looked at the bottom-up algorithm,

which starts the construction of types from the "bottom" of the component dependency tree instead of recursively from the top. This has the advantages of being conceptually simpler, as well as requiring constant-size stack space.

9.2 Further work

9.2.1 Language extensions

The implementation of the type system has been based on [2], which is still ongoing work. The authors are currently planning additional features to the language to make it more useful in more situations. In anticipation of this, a goal in the implementation stage has been to keep the OCaml code easy to read, understand, and extend.

One of the features being worked on is parallel composition, represented by component expressions on the form $(A||B)$, which means A and B will be executed in parallel. We have seen experimental support for this in our implementation, but the typing system is still in need of further refinement to keep the instance bounds strict. When the theory is ready, it should be simple to update the type inference implementation to reflect the refined theory.

9.2.2 Development of the implementation

A natural extension of the type system implementation would be to adopt the bounds checking algorithm. The efforts involved in this is not substantial, and essentially involves extending the syntax for declarations to include the upper bounds for the component being declared, as well as adding the logic to check that sequential composition does not result in instance counts exceeding boundaries.

Alternatively, the implementation may be changed to the bottom-up algorithm. As this approach checks the boundaries only at the end, this has the advantage that one can allow bounds checking to be disabled or enabled according to the needs of the tool or composer.

The tools can also be enhanced. For the `comptop` tool, it would be useful to have more versatile ways to manipulate the declaration. Most importantly, support for removing individual components, or changing the declaration of a component, should be of priority.

9.2.3 Applications

The component language and type system are not strictly limited to applications within software component contexts. An interesting application would be the modelling of conventional object-oriented program code. This could be useful as a way to measure resource demand by class instantiation and function invocation. The idea is to work on the function level by transforming a function body according to some rules:

$$\begin{aligned}
\Phi(\textbf{new x}) &= \texttt{new}\, x \\
\Phi(\textbf{if}\ b\ \textbf{then}\ P_1\ \textbf{else}\ P_2) &= (P_1 + P_2) \\
\Phi(f(x)) &= \{\Phi(x)\}
\end{aligned}$$

Sequential composition is preserved, and what remains after transformation is deleted. The result is a component expression which is subject to type checking for resource management. The challenges here is finding a way to model loop statements and transforming threaded execution into parallel expressions.

Bibliography

[1] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[2] Marc Bezem and Hoang Truong. Counting Instances of Software Components. In *ICALP - LICS 2004: Workshop on Logics for Resources, Processes and Programs*, 2004.

[3] Dietrich Birngruber. A Software Composition Language and its Implementation. *Lecture Notes In Computer Science*, 2244:519–529, 2001.

[4] Martin Büchi and Wolfgang Weck. A Plea for Grey-Box Components. In *Foundations of Component-based Systems '97*, 1997.

[5] Luca Cardelli. *Handbook of Computer Science and Engineering*, chapter 103. Digital Equipment Corporation, 1997.

[6] William D. Clinger. Proper Tail Recursion and Space Efficiency. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1998.

[7] Robert Englander. *Developing Java Beans*. O'Reilly, 1995.

[8] Clemens Szyperski et al. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Pub Co, second edition, 2002.

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

[10] Hoang Lam and Thuan L. Thai. *.NET Framework Essentials*. O'Reilly, third edition, 2003.

[11] Markus Lumpe. *A π-Calculus Based Approach for Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, 1999.

[12] M. Douglas McIlroy. Mass Produced Software Components. Website. `http://www.cs.dartmouth.edu/~doug/components.txt`.

[13] Eric Meijer and John Gough. Technical Overview of the Common Language Runtime. Microsoft, 2000.

[14] Erik Meijer and Clemens Szyperski. Overcoming Independent Extensibility Challenges. *Communications of the ACM*, 45(10):41–44, 2002.

[15] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, 1990.

[16] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, second edition, 2000.

[17] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Langages for Concurrent Systems*, volume 924, pages 147–161. Springer-Verlag, 1995.

[18] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[19] João Costa Seco and Luís Caires. A Basic Model of Typed Components. In *European Conference on Object-oriented Programming*. Springer-Verlag, 2000.

[20] Niklaus Wirth. From Modula to Oberon. *Software Practice and Experience*, 18(7), 1988.