The University of New South Wales
School of Computer Science and Engineering

A .NET Bridge for Haskell:
Dancing with the Devil

Andrew Appleyard

Bachelor of Science (Computer Science)
October 2007

Supervisor: Manuel M. T. Chakravarty
Assessor: Gabriele Keller

Abstract

Libraries are essential for software development in any language. Access to the extensive collection of high-quality libraries provided by the Microsoft .NET Framework is, understandably, something that many programmers require. This thesis addresses the challenge of providing access to .NET libraries from Haskell by developing a runtime bridge, called Salsa, between their respective runtime systems. In doing this, a new technique for binding object-oriented subtyping and method overloading in Haskell was developed, which is type safe and has a convenient syntax.

Acknowledgements

I spent many hours at one with my computer while working on this thesis, but to think that I did this work alone would be far from the truth. Many people have helped me in the months I spent producing this thesis, and I owe my thanks to you all. I would especially like to thank the following people:

To Simon Winwood for introducing me to the wonderful Programming Languages and Systems (PLS) research group at UNSW. It is such an amicable bunch of intelligent, fun-filled people, and I could not imagine a better place to do my thesis. My thanks go out to everyone in the group for your help and encouragement.

To Manuel Chakravarty for being such an outstanding supervisor. Your unbridled enthusiasm and charisma has made this work exciting far beyond my expectations.

To André Pang, who has gone out of his way to help me with my work, and whose thesis serves as a basis for much of mine.

To my mother, for looking after me as she always has, and for making my life much easier than it would otherwise have been so that I could concentrate on this thesis.

And to my father, for instilling in me a love of programming and of beautiful code when I was young. This love continues to this day.

Contents

1

Introduction

Libraries are essential for software development in any language. In many cases the choice of libraries dictates the choice of programming language. This is unfortunate since it excludes great languages that would otherwise be a perfect tool for the job.

Haskell is a purely functional, lazy, general-purpose programming language with a sophisticated type system, high-quality native-code compilers, and elegant libraries. It is an attractive choice for many programming tasks, especially since it allows the programmer to work at a high level of abstraction.

Microsoft's .NET Framework [3] is an object-oriented software platform that includes, among other features, a virtual-machine runtime environment and an extensive collection of high-quality libraries for a number of domains. It is essential for writing modern applications on the Windows platform, as well as in organisations with existing investments in Microsoft technology.

Can we have the benefits of both Haskell and .NET? The ability to use .NET libraries from Haskell is a particularly attractive combination. Unfortunately, the Haskell and .NET worlds are far apart, and despite earlier attempts at interoperability, no satisfactory solution has been developed to date.

This thesis shows that it is possible to have the benefits of both Haskell and .NET, by developing a runtime bridge between the two systems that allows them to interoperate. The bridge, called Salsa, focuses on providing convenient and natural access to .NET libraries from Haskell. The difference between Haskell and .NET, with respect to their runtime environments and type systems, makes this a challenging implementation and research task.

Beyond describing the design and implementation of the bridge, and how it addresses the challenges involved, this thesis presents new techniques for binding object-oriented concepts in Haskell. The use of labels to bind .NET types and members, is described; as is the use of type-level algorithms for supporting subtype polymorphism and method overloading. Both of these techniques allow for type-safe access to object-oriented concepts with a natural syntax.

Overview

Chapter 2 outlines the goals of this thesis in order to accurately describe the scope of the work.

Chapter 3 describes relevant background material, including type families, a recent extension to the Haskell type system that is employed in this work.

Chapter 4 describes the various problems that arise in building a bridge that satisfies the goals outlined in Chapter 2.

Chapter 5 analyses existing solutions and other related work, noting how they differ from, or support, this work.

Chapter 6 gives an overview of the design and implementation of the bridge, including how it addresses the challenges described Chapter 4.

Chapter 7 shows how labels can be used to map .NET types and their members into Haskell with a natural syntax.

Chapter 8 describes a technique for performing overload resolution and implicit conversions in the bridge, by implementing type-level algorithms using type functions.

Chapter 9 demonstrates how the bridge can be used to interoperate with .NET libraries by presenting a small sample program.

Chapter 10 concludes this thesis and gives directions for future work.

2

Goals

The goal of this thesis is to allow Haskell and .NET programs to interoperate by building a software bridge between the Haskell runtime system and .NET execution engine. The focus of the bridge is to provide convenient access to .NET libraries *from* Haskell, but it must also provide sufficient access to Haskell from .NET so that typical .NET libraries can be used.

There are a number of ways of going about developing such a bridge. Design decisions and compromises must be made. To guide the making of these decisions, a number of aspirations for the work are described below. These aspirations are also useful for scoping the work and for evaluating existing work.

Usability .NET libraries should be relatively natural and easy to use from Haskell. The benchmark for this goal is C#; the closer the bridge approximates C# in this respect, the better. Bindings are typically forced to make compromises in this space because of the mismatch between functional and object-oriented programming (especially with respect to their type systems).

Safety Incorrect use of the bridge itself should not be able to cause the Haskell program to crash (violate run-time safety). Ideally, the bridge should enforce the same static checks that a typical .NET compiler would enforce. Again, C# can be used as a benchmark.

Convenience Users of the bridge should not have to write boilerplate code in order to use .NET libraries.

Lightweight The system should not impose heavy changes to the way that Haskell programs are developed and executed. This excludes making major extensions to the language and any retargeting of the compiler.

Practicality The system should be useful in real-world contexts. This implies that the bridge should support a sufficient feature set and be reasonably straightforward to deploy.

Ultimately the goal is to allow anyone to take advantage of the benefits of Haskell and the benefits of .NET when writing applications, without throwing away the existing investment in native Haskell implementations and libraries.

Non-goals of the project

Why not compile Haskell code directly to the .NET Common Intermediate Language (CIL)? Retargeting a Haskell compiler so that it generates CIL is an obvious approach to solving the issue of Haskell and .NET interoperability. Haskell code would then run directly on the .NET runtime, and the Haskell runtime system would become the .NET runtime, eliminating the need for communication between the two. There are a number of reasons however, that make this approach an inappropriate choice for this project. They include:

Enormity of the task Retargeting a native-code Haskell compiler to a stack-based virtual machine is a large and difficult task. This is evident in the way O'Boyle's honours thesis [19] attempts the task, which uses three different compilers (GHC, the Mondrian compiler, and a C# compiler) as well as a custom translator in order to compile Haskell code to CIL.

Unacceptable performance The design of .NET and its execution engine is clearly focused on strict, statically typed, object-oriented languages. Producing efficient .NET code for a typical Haskell program (employing lazy evaluation and higher-order functions), is well beyond the scope of an undergraduate thesis.

Heavyweight approach One of the goals of this project is to provide a *lightweight* means of accessing .NET libraries for real-world Haskell programs. Retargeting a full-featured Haskell compiler is not generally considered a lightweight approach.

More importantly, the retargeting approach only solves part of the problem. The low-level issue of runtime system interoperation is eliminated, but the challenge of mapping between Haskell and .NET concepts remains. Retargeting is simply too much work for little gain. Providing the interoperability in the form of a runtime bridge is therefore a more practical solution given the goals of the project.

3

Background

3.1 .NET and C#

Some background knowledge of both .NET and C# is required to fully appreciate this thesis. Given the task of interoperability with .NET, it is clear that some knowledge of .NET is required. In many cases however, the bridge must deal with issues that the .NET specification defers to the language designer. Where a concrete language is required, we use C# because of its very close correspondence with .NET's type and object system. In fact, .NET and C# are essentially interchangeable in the context of this work.

3.1.1 Definitions

C# and the .NET framework carry with them a substantial collection of acronyms and new terms. Definitions of relevant acronyms and terms in the context of this thesis are given below for the benefit of readers who are familiar with object-oriented systems, but not with .NET in particular.

.NET execution engine: the .NET virtual-machine.

Common Language Runtime (CLR): Microsoft's implementation of the .NET execution engine.

Mono: an open-source implementation of the .NET execution engine.

Common Intermediate Language (CIL): the specification of .NET byte-code.

Managed code: code that is executed by the .NET execution engine.

Assembly: a container of .NET code and associated metadata. Similar to a shared library or executable file.

Reference type: a .NET type that is stored on the heap and always accessed via an object reference. Reference types do not derive from System.ValueType.

Value type: a .NET type that is stored on the stack or inside the values of other types. Value types derive from System.ValueType. A value of a value type can be *boxed* in order to obtain a reference to (a copy of) the value.

3.1.2 Scope limitations

Comfortingly, a complete understanding of the intricacies of .NET and C# is not required for this work. Many aspects of .NET are irrelevant because our goal only involves *accessing* .NET libraries, and not *creating* new .NET libraries. Some noteworthy features of .NET and C# that can be overlooked include:

Type and member visibility Only public types and public members of such types are visible when accessing a .NET library through the bridge. As such, modifiers pertaining to *visibility* ('public', 'protected', 'private' and 'internal') are not covered within.

Structures The bridge marshals all non-primitive values (such as C# *structs*) by reference because Haskell cannot, in general, directly access the data contained in these values. From the point of view of bridge's type system, structs can thus be treated as classes.

Static constructors In contrast to instance constructors, static constructors cannot be called by the programmer and do not affect the bridge.

Some other features have been removed purely to simplify the exposition, these include:

- Array types
- Unsigned primitive types
- Reference and output parameters

3.2 Type systems and inference rules

Some chapters of this thesis formally specify select parts of the .NET type system and the bridge implementation. This includes defining subtyping relations and certain sets (of types). Inference rules are used to describe these relations and sets. In order to make this thesis self-contained, the concept of inference rules are explained below. (For a thorough treatment of the theory of type systems, refer to Pierce [23]).

An inference rule, such as the following:

$$\frac{p_1 \qquad p_2 \qquad p_3}{c}$$

states that if all the logical statements above the line (the premises p_1, p_2 and p_3) hold, then one can infer that the logical statement below the line (the conclusion c) also holds. If there are no statements above the line, the conclusion always holds (it is an axiom).

Collections of inference rules are used to define sets and relations. The set or relation defined by the collection of inference rules is the smallest set that contains every element that can be derived from the inference rules (starting with the axioms).

As an example, consider the following two inference rules that define the set of natural numbers, 'nat':

$$\frac{}{0 \text{ nat}} \quad (\textsc{Zero}) \qquad \frac{n \text{ nat}}{\mathrm{s}(n) \text{ nat}} \quad (\textsc{Succ})$$

The axiom (Zero) states that '0' is a natural number (i.e. a member of 'nat'). The rule (Succ) states that if you have a natural number n, then the successor of that number, s(n), is also a natural number.

In the case of the typing rules discussed in this thesis, we typically wish to devise an algorithm that can deterministically decide if a particular element is in the particular set or relation. Here the rules serve as a concise and accurate notation for specifying what the algorithm is to achieve.

3.3 Type families

Haskell's *type classes* support the ad-hoc overloading of functions, or equivalently, the definition of *type-indexed* functions. This allows a single function name to be associated with any number of definitions, with the appropriate definition being selected based on one (or more[1]) of the types involved in the function application.

Type families [1, 26] extend this overloading support into the realm of types by providing the ability to define *type-indexed types* (for both data types and type synonyms). There are three ways of defining type-indexed types with type families:

Associated data types A data type defined in a type class, with definitions given by instances of the type class.

Associated type synonyms A type synonym defined in a type class, with definitions given by instances of the type class.

Stand-alone type functions A type synonym defined at the top-level (using the **family** keyword), whose definitions are given by top-level type instance declarations.

Type-level computation

Associated type synonyms and stand-alone type functions pave the way for performing computations at the level of types in Haskell. Type functions, in particular, provide a convenient syntax for defining these type level computations; and make the definition of type level functions look much like the pattern-based definitions of standard Haskell functions.

The rest of this section defines some primitive type functions for performing type-level computations, which are built on later in this thesis. For clarity, and to avoid name clashes, type functions defined solely for performing type-level computations are prefixed with the letter 'T'.

Performing computation at the type-level in Haskell is not new. Prior to the existence of type families, functional dependencies [9] have been used to perform such computations. The basic definitions for type-level booleans, logic operations, and lists that follow are also defined in Kiselyov, Lämmel and Schupke's work on HList [14], for example.

[1]This feature requires the multi-parameter type classes extension.

Type-level booleans

In Listing 3.1, we define constructor-less data types to represent the boolean values of true and false; these are the type-level boolean literals. Type-level boolean logic functions can be defined in terms of these type-level booleans. Listing 3.2 defines short-circuited conjunction, disjunction and negation at the type-level. Finally we define a type-level conditional, for making decisions based on the type-level boolean values, in Listing 3.3.

Type-level lists

Type-level list processing is used extensively in Chapter 8. We define constructs for building type-level lists in Listing 3.4. A constructor-less data type TNil represents the empty list, while the infix type constructor :::[2] is our list constructor (cons). The listing also contains a fixity declaration for ::: so lists can be written without extraneous parentheses.

By default, only a restricted form of type function definition is allowed in order to ensure that type-checking will terminate. Many interesting type-level computations however cannot, in general, be proven to terminate. In order to support these computations we instruct the compiler to relax termination checking[3].

[2]Use of infix type constructors requires the type operators extension.

[3]In the Glasgow Haskell Compiler, this option is called 'allow undecidable instances'.

```
data TTrue
data TFalse
```

Listing 3.1: Type-level boolean literals

```
type family   TAnd x y
type instance TAnd TFalse x = TFalse
type instance TAnd TTrue  x = x
type family   TOr x       y
type instance TOr TTrue  x = TTrue
type instance TOr TFalse x = x
type family   TNot x
type instance TNot TTrue   = TFalse
type instance TNot TFalse  = TTrue
```

Listing 3.2: Type-level boolean logic functions

```
type family   TIf c      a b
type instance TIf TTrue  a b = a
type instance TIf TFalse a b = b
```

Listing 3.3: Type-level conditional function

```
data TNil
data x ::: xs
infixr 5 :::
```

Listing 3.4: Type-level list construction

4

Bridges for Interoperability

Haskell and .NET are worlds apart. This is why a software bridge is required to provide interoperability between them in the first place. Bridging the two systems is difficult for two main reasons:

1. Both Haskell and .NET have their own runtime systems that perform tasks such as memory management and threading. By design these runtime systems only manage their own code and data. Sharing code and data between them requires extra effort.
2. There is a mismatch between Haskell's concepts and the object-oriented concepts of .NET. This is manifested clearly in their different type systems, and how they assign types to data and code. Encoding .NET concepts in Haskell in a natural way, and vice versa, is difficult.

This leads to the two core tasks that the software bridge must perform: *runtime system interoperation*; and *type and object system mapping.* The sections below expand on these tasks.

4.1 Runtime system interoperability

Providing the required infrastructure so that the .NET execution engine and the Haskell runtime system can interoperate is the core low-level task that must be addressed by the bridge implementation. There are a number of issues involved in carrying out this task:

Hosting To attain the required level of interoperability, the respective runtime systems must be loaded into the same process. In the case of our bridge, either the Haskell program can load the .NET execution engine into its process, or vice versa.

Memory and reference management Both .NET and Haskell use a garbage collected memory management scheme. A mechanism is required to prevent references to objects in the foreign runtime from being prematurely garbage collected.

Data marshaling Haskell and .NET store their data in different memory locations and using different formats. Where data must be copied from one runtime to the other, a mechanism is required to convert the data into an acceptable format for the receiver.

Transfer of control Calling a .NET method from Haskell, or a Haskell function from .NET, requires transferring the thread of control from one runtime to the other.

Threading .NET allows the use of multiple operating system threads and Haskell allows both lightweight and operating system threads to be used. Consequently, each runtime must be re-entrant, and the bridge must handle concurrent access to any of its global data structures.

Techniques for dealing with many of these issues are described in various interoperability papers [6, 7, 16, 18]. Notably, Haskell's excellent Foreign Function Interface (FFI) allows for relatively elegant and straightforward solutions to some of these issues. By applying existing interoperability techniques, the Haskell FFI, and the appropriate .NET APIs, the issues outlined above can be solved by careful programming.

4.2 Type and object system mapping

Runtime system interoperation is sufficient for a bridge in the same way that a Turing machine is sufficient for arbitrary computation; however Haskell and .NET have high-level concepts that we would like to maintain (as much as possible) when using the bridge. A type and object system mapping between Haskell and .NET allows us to do this by exposing .NET concepts in Haskell and Haskell concepts in .NET. Such a mapping is designed to satisfy the goals of *usability* (it must feel natural to the programmer) and *convenience* (it should be automatic).

The features of Haskell's type system and those of object-oriented systems (like .NET) are known to not play well together. This is evident in the number of papers covering the topic, and implies that creating a type and object system mapping that satisfies the goals of the bridge is a worthwhile challenge.

Since accessing .NET libraries from Haskell is the focus of this work, we will concentrate on mapping .NET concepts *into* Haskell. There are a number of difficulties involved with this:

- Subtyping
- Method overloading
- Namespaces
- Generics

These difficulties are described in more detail below.

4.2.1 Subtyping

The classes in .NET, as in most object-oriented languages and systems, form an inheritance hierarchy. A particular .NET class can only *derive* from one superclass (single, implementation inheritance) but can *implement* any number of interfaces (multiple, interface inheritance). A subtyping relation is formed over the types of classes and interfaces: one class or interface is a subtype of another if it derives or implements the other (either directly or indirectly).

Haskell has no concept of subtyping. This poses a difficulty to exposing .NET libraries in Haskell. These libraries are designed with subtyping in mind, and, for example, expect the caller to be able to pass any subtype of the type of a formal argument, for the actual argument. At the very least, a way of coercing types up and down the hierarchy from Haskell is necessary. Without implicit coercions on method calls however, the resulting code will contain a number of superfluous casts, and look unnatural.

4.2.2 Method overloading

A single method name may be associated with different underlying method implementations in .NET, with the underlying implementation being resolved statically by the compiler. The different implementations for a method can be found in different classes or even in the same class. In the former case, the target of the method call is used to determine the class that contains the method, while the types of the method's arguments are used to resolve the method in the latter case.

The process of resolving a method call to the appropriate method implementation is controlled by a set of rules in the compiler. The *C# Language Specification* [4, Section 14.4.2] includes a number of sections stating the rules of "overload resolution" in C#. The presence of subtyping makes these rules rather complicated.

In order to call .NET methods with overloaded names, in a natural way, from Haskell, a system of resolving the appropriate method is required. Pang and Chakravarty [21] describe a method of resolving overloading using Haskell's type system (in particular, multi-parameter type classes with functional dependencies). Other methods include using mangled method names, or resolving the overloading at runtime instead (provided the required static typing information is available).

4.2.3 Namespaces

Nested namespaces are supported by .NET and used extensively in its libraries. Explicit namespaces can be created, but classes and structures are also (implicitly) namespaces. Any kind of type definition, such as a class declaration or enumerated type, can be made in a namespace. This implies that classes can be nested inside each other (although this usage is not encouraged for the external interfaces of .NET libraries).

Haskell includes a simple module system, which allows for hierarchically *named* modules, but ultimately functions and types end up being imported into a flat namespace.

Without some support for namespaces in the bridge, Haskell programs will need to use fully qualified names for .NET classes, methods and other items. The resulting code would be unnatural to read and to write, so some solution to the problem must be provided by the bridge.

4.2.4 Generics

With version 2.0 of the .NET framework, Microsoft added support for parametric polymorphism (generics) to the .NET runtime and languages. Both .NET types and .NET methods can be parameterised by one or more types (refer to Kennedy and Syme [12] for details).

Much of the research on Haskell interoperating with object-oriented systems was performed before generics support was introduced into the mainstream object-oriented frameworks (Java and .NET). Consequently, little research has been conducted on the mapping of object-oriented generic types into Haskell.

5

Related Work

In this section, existing solutions to the challenge of interoperating with .NET from Haskell are examined, along with other systems that solve a similar problem, and papers that cover relevant research areas.

5.1 Existing solutions

There are not many runtime bridges between Haskell and .NET. The most prominent, and useful, of these bridges is Hugs98 for .NET, which we will examine here. Another runtime bridge, called GHC.NET is part of the Glorious Haskell Compiler (GHC) but it is, at the time of writing, suffering from bit-rot.

5.1.1 Hugs98 for .NET

Hugs [10] is an interpreter for Haskell, written in C, by Mark P. Jones. It was later extended by Sigbjorne Finne, to create *Hugs98 for .NET* [5]. The extended version provides support for interoperating with the .NET framework. Unfortunately, no papers could be found that describe Hugs98 for .NET, so the following information was obtained by perusing the Hugs98 documentation and source code.

The interoperability provided by Hugs98 for .NET is mostly unidirectional, allowing access to .NET *from* Haskell. There are two ways to access .NET: by declaring foreign imports (using the 'dotnet' calling conversion); or by calling functions provided in the 'Dotnet' module. Both allow access to .NET class members, including: invoking methods and constructors; and getting and setting the values of fields and properties.

There is experimental support for accessing Haskell from .NET. This is done by wrapping Haskell functions as .NET delegates (or as .NET classes) dynamically.

Since this occurs at run-time it is not possible for external .NET code, such as a C# program, to compile against such classes; they can only use them at run-time.

Implementation details

In order to interact with the .NET framework and its runtime engine, Hugs98 for .NET uses Microsoft's *Managed Extensions for C++*. Managed Extensions for C++[1] is an extension to C++, implemented by the Microsoft Visual C++ compiler, that allows native code to interoperate easily with managed (i.e. .NET) code. The entire Hugs code base, including the Hugs98-for-.NET extensions, is compiled with this compiler to produce Hugs98 for .NET. The result is a mixed-mode executable (one containing both native and managed code) that automatically uses the CLR to execute its (managed) code.

Requests to access .NET from Haskell code (either via foreign imports, or via the Dotnet module) result in calls to a handful of *primitive* functions implemented in C++. These functions unmarshal Hugs values to their C equivalents (as necessary), and then use methods in the .NET reflection library (System.Reflection) to perform the desired .NET operations. The C++ compiler makes this possible (the .NET methods are accessible directly from the C++ code).

The experimental support for calling Haskell functions from .NET (via delegates and dynamically generated classes) is implemented in a C++ class. This class is compiled and linked into the mixed-mode Hugs executable and then accessed via the bridge itself. The class uses Reflection.Emit (a library in .NET for generating CIL code dynamically) in order to construct .NET delegates/classes that wrap Haskell functions. These delegates/classes ultimately call functions defined in the Hugs runtime to invoke the desired Haskell functions. Again the C++ compiler makes this possible (the native functions in the Hugs runtime are directly accessible from the .NET class).

Mapping details

All references to .NET objects are represented in Hugs98 for .NET using the `Object a` abstract data type. This data type accepts an unused type parameter, allowing the use of *phantom types* [7] to encode .NET subtyping (with some limitations). Values of the `Object` type along with any other types that are members of the **NetType** type class are allowed to be passed to, and returned from, .NET calls. Beyond this, the Hugs extension does not attempt to map

[1]Managed Extensions for C++ is superseded by C++/CLI.

.NET types into Haskell. This job is left to the wrapper generator (hswrapgen).

The wrapper generator uses .NET reflection to generate a Haskell module (.hs file) that includes the foreign import declarations and data type declarations that are required to use a particular .NET class. It produces an empty data type declaration (with a phantom type parameter) and a type synonym (indicating that it is a subtype of its parent class) for the class:

```
data DateTime_ a
type DateTime  a = ValueType (DateTime_ a)
```

(which indicates that the .NET class DateTime is a subtype of ValueType)

It also produces foreign import declarations for all of the constructors, methods (including property get and set methods) and fields. No special syntax is provided for accessing properties (unlike C#). For example, the read-only Month property of the DateTime class is accessed directly via its get method:

```
foreign import dotnet "method System.DateTime.get_Month"
  get_Month :: DateTime obj → IO Int
```

The generator appends a numeric suffix to any function names that are required more than once. This occurs when a method or constructor is overloaded.

Limitations

Some of the limitations of Hugs98 for .NET are outlined below:

Basic type mapping The type mapping performed by the wrapper generator has a number of limitations that make foreign libraries inconvenient to use from Haskell:

- The names of overloaded methods are mangled (suffixed with a number) which makes writing the code difficult.
- Names common to more than one class must be resolved through the Haskell module system, thus littering the code with qualified names.
- Interfaces are not included in the subtyping hierarchy due to the use of phantom types (which only support single inheritance).

These limitations ultimately make the resulting code unnatural to deal with, both for Haskell and .NET programmers.

No support for .NET 2.0 The bridge operates only with versions 1.0 and 1.1 of the .NET framework. This rules out access to a large number of .NET libraries, including the latest .NET 3.0[2] libraries. There are significant differences between .NET 1.0/1.1 and .NET 2.0, principally due to the introduction of generics (parametric polymorphism) in the latter.

Implementation portability The bridge works only on a specially compiled version of the Hugs interpreter, and then only on Microsoft Windows.

Platform portability The use of Managed Extensions for C++ to implement the bridge means that the .NET runtime must support mixed-mode executables. Mono does not support these (and, to my knowledge, there are no plans to add support for them).

Limited bidirectional bridge support Even when a bridge is focused on interoperability in a particular direction (from Haskell to .NET, for example), sufficient support must be provided in the other direction to support call-back functions. Hugs98 for .NET supports exposing Haskell functions to .NET in the form of EventHandler[3]-typed delegates. This works well for Windows Forms 1.0 (an early GUI library for .NET), but it is not sufficiently general to support all library call-backs. Ideally, any Haskell function having a matching .NET delegate type should be able to be exposed to .NET.

Efficiency The bridge uses a reflection technique that is particularly slow when making calls into .NET. The InvokeMember method of the .NET `Type` class used by Hugs98 for .NET is the "slowest of the late-bound invocation mechanisms" [24]. Since Hugs itself is not a performance-focused implementation of Haskell, in context this issue is minor.

Some of these limitations, such as the use of Managed Extensions for C++ are inherent in the design and implementation of the bridge. So modifying Hugs98 for .NET to resolve these limitations is not a feasible option.

[2]The (confusingly named) *.NET Framework 3.0* is a collection of .NET libraries that run on top of version 2.0 of the .NET Framework.

[3]An EventHandler delegate represents method of the form: *void f(object sender, EventArgs e)*.

5.2 Other systems

In this section we examine other systems that deal with interoperability, either with Haskell, or with .NET. None of the systems described here aim to solve the same problem as this thesis, but the problems that they do solve are nonetheless relevant to this work.

5.2.1 MochΛ

André Pang's thesis [20] and the associated paper *Interfacing Haskell with Object-Oriented Languages* by Pang and Chakravarty [21] both describe a general system for interoperability between Haskell and object-oriented component systems. The work builds on Lambada [18] and the techniques described by Shields and Peyton Jones [28]. From here on we will only refer to the paper, but everything discussed applies to the thesis as well. The work makes three main contributions:

- A system for dealing with object-oriented method overloading in Haskell, that makes use of multi-parameter type classes and functional dependencies.
- Using compile-time meta-programming in Haskell (Template Haskell) and reflection to automatically generate interface bindings for an external library without using external tools.
- A Haskell to Objective-C [8] binding called Mocha that applies the above contributions in a real bridge implementation.

Pedagogically, the work is similar to this thesis. Both are concerned with getting Haskell to interoperate with an external object-oriented system in a convenient and natural way. The main difference between the two is the foreign system being interoperated with. Although Pang's research describes a relatively general object-oriented mapping, the implementation (MochΛ), ultimately interoperates with Objective-C as opposed to .NET.

There are some significant differences between Objective-C and .NET that affect the way a bridge to Haskell would be implemented (with respect to both runtime system interoperability and type mapping). These differences are summarised in Figure 5.1.

Since this thesis is explicitly focused on .NET, the type and object system mapping effort is tailored specifically to .NET rather than object-oriented component systems in general. This means that .NET concepts like delegates, properties and

	Objective-C	**.NET**
Object interaction	Dynamic message passing	Static class-based method calling
Memory management	Reference counted	Garbage collected
Nested namespaces	Not supported	Supported (and used extensively)

Figure 5.1: Comparing Objective-C and .NET

enumerated types will have a 'first class' status in the mapping and will be more natural to use in Haskell than they would otherwise be.

Despite the differences between the paper and this thesis, it is clear that the contributions of the paper are relevant here. We will now examine how Pang and Chakravarty deal with mapping subtyping and method overloading into Haskell. Both of these are highlighted, in §4.2 as challenges that must be addressed by the bridge.

Mapping subtyping

This section outlines the subtype mapping system that is described in the paper. For each object-oriented class, a Haskell data type and type class is created. For example, if we have a Shape and a Circle class, we get:

```
data Shape
class SubShape a

data Circle
class SubCircle a
```

Then for each class, instances are declared in the type classes that correspond to its class or superclasses. Continuing our previous example, if Circle is a subclass of Shape, we get:

```
instance SubShape Shape
instance SubShape Circle
instance SubCircle Circle
```

The type classes can then be used in the context of type expressions for objects (class instances). Continuing our example, ($SubShape\ a \Rightarrow a$) is the type associated with the Shape class (or one of its descendants) and ($SubCircle\ a \Rightarrow a$) is the type associated with the Circle class (or one of its descendants).

Mapping overloaded methods

The issue of method overloading is addressed in the paper by using multi-parameter type classes and functional dependencies. A number of variations are covered; a simplified version of one such variation is described here.

For each unique method name define a type class and a function. For example, if we have a Console class, with an overloaded method called Write, we get:

```
class Write target arguments result | target arguments → result

write :: (Write target arguments result) ⇒
         target → arguments → result
write = ...
```

Notice that the type class accepts three parameters: one for the type of the target object, one for the type of a tuple holding the method's arguments, and one for the result type. Also notice that a functional dependency, indicating that the type of the result is functionally dependent on the type of the target and arguments, is included in the class definition.

Then for each overloaded version of the method, define a corresponding instance of the type class. Assuming that there are two versions of Write in the Console class, one taking an Integer, the other a Bool, then we have:

```
instance Write Console Integer (IO ())
instance Write Console Bool    (IO ())
```

The **write** function can now be used to call either of the overloaded Write methods in the Console class.

Unfortunately this system breaks down when subtyping and method overloading are combined. Wolfgang Thaller [30] discovered this problem while attempting to use Mocha's type mapping system in HOC (another Haskell to Objective-C bridge). The problem is exhibited when methods from two unrelated classes have the same name and argument types. For example, if a Console class and an Author class have a Write method accepting a string, the mapping gives the following instances:

```
instance (SubConsole target) ⇒ Write target String (IO ())
instance (SubAuthor  target) ⇒ Write target String (IO ())
```

Notice that, due to the use of subtyping, the target object type has moved from the head of the instance declaration to the context. Since the compiler only looks at the head of the instances when matching them:

> **instance** *Write target String* (*IO* ())
> **instance** *Write target String* (*IO* ())

it has no way of telling them apart, and raises a compilation error.

Conclusion

The work of Pang's thesis and the associated paper serve as a basis for the work in this thesis, especially with respect to the type and object system mapping. Overcoming the limitations of the work, and determining how to incorporate namespaces and other .NET concepts into it, constitutes much of the research undertaken herein.

5.2.2 Lambada

Lambada [18] is a runtime bridge, created by Meijer and Finne, that provides interoperability between Haskell and Java.

The approaches to runtime system interoperability taken by Lambada and by Salsa are similar. Both make use of Haskell's FFI to communicate with the foreign runtime system, and both operate as a Haskell library that does not require any special language extensions. Obviously the two bridges differ significantly with respect to the foreign runtime being interoperated with (despite the similarities between Java and .NET, each have different ways of interoperating with their runtime systems).

For the type mapping Lambada uses a hybrid approach to deal with subtyping. Phantom types are used to encode Java's class hierarchy, while type classes are used to represent Java interfaces. Lambada does not attempt to deal with the other challenges outlined in §4.2 (method overloading, namespaces and generics).

Lambada employs a tool called H/Direct to generate the Haskell bindings for Java classes. This works in a rather indirect way, requiring the Java classes first be described in Interface Description Language (IDL) before being converted into Haskell code by H/Direct.

5.2.3 F#

F# is an OCaml-inspired variant of ML that compiles to .NET CIL and runs directly on the .NET platform. It was written by Don Syme at Microsoft Research, and continues to be developed there. The F# website [25] describes it as a multi-paradigm language, since it naturally supports the imperative, functional, and object-oriented programming styles. Of particular relevance to this thesis is that F# is a functional language, with type inference, that happens to have excellent bidirectional interoperability with .NET.

Since the interoperability technique being applied in this thesis (runtime bridging) and that of F# (compilation) are very different, it would be unrealistic to expect the bridge to have the same level of .NET interoperability as F#. However as a standard to strive for, and as an example of how to adapt a functional language to fit the .NET mould, F# is certainly worthy of study.

Although F# is based on the OCaml language, it does not aim to be an OCaml implementation and there are significant syntactic and semantic differences between the two. These differences, according to the F# website, arise "from essentially unavoidable changes for the design of a .NET language." Since making significant changes to the Haskell language is prohibited by the goals of this project, therein lays a significant challenge. How can we maximise ease of interoperability between Haskell and .NET without significantly changing the language?

We will now examine some of the areas where F# differs from OCaml. The main differences are in the following areas:

Subtyping Whether one type is a subtype of another can be defined explicitly using their names (*nominal subtyping*) or implicitly in the structure of the types involved (*structural subtyping*). OCaml uses structural subtyping (like many other languages in the research community), while .NET employs nominal subtyping (which is typical of most object-oriented languages and systems). Since Haskell does not have subtyping at all, the choice for the bridge is either to model a form of subtyping in Haskell, or have the bridge absorb it (using dynamic checks) and hide .NET's subtyping altogether from the Haskell side (or perhaps something in between these two extremes).

Functors The concept of parameterised modules (functors) is not present in either .NET or Haskell, so F#'s lack of functors does not concern us.

F#, like Haskell, uses type inference to free the programmer from annotating every expression with a type. The algorithm used for type inference in F# plays are large role in making .NET libraries naturally accessible from F# (outward interoperability). In particular, it attacks the problem arising from mixing method overloading and subtyping (described in §4.2.2) by:

- Implicitly coercing (narrowing) the type of actual arguments to the formal argument types. This allows subtypes of the actual argument types to be passed to methods.
- Requiring explicit type annotations on the arguments of some calls to overloaded methods. This is necessary to resolve the overloading when it is ambiguous.
- Using a complicated set of rules to deal with overload resolution.

5.3 Relevant papers

In this section we describe two noteworthy papers that are relevant to the type and object system mapping aspect of this work.

5.3.1 Object-oriented style overloading for Haskell

The premise of the paper *Object-Oriented Style Overloading for Haskell* [28], by Shields and Peyton Jones, is the desire to use object-oriented libraries, like those of .NET and Java, from Haskell. It focuses not on the technical aspects of such interoperability, but on mapping the type systems of these object-oriented systems into Haskell's type system. It is one of the seminal papers in this area.

The paper describes a number of alternative solutions to the challenge of subtyping, they include:

1. Using phantom types
2. Mapping each class to a type class
3. Mapping each class to a data type and then encoding subtype relationships with type classes
4. "full-blown subtype constraints" [28]

Only the last alternative requires changes to Haskell's type system. Shields and Peyton Jones conclude that the third alternative is sufficient (this is the same technique that MochΛ employs to deal with subtyping, as described in §5.2.1).

The other challenges, method overloading and namespaces, are addressed by providing a way to overload method names in Haskell. By overloading across the target class of the method, the namespace issue is resolved; and by overloading across the argument types of a method, the method overloading issue is resolved. The authors note that this resembles the multi-methods of the Common Lisp Object System (CLOS).

To allow for the method name overloading, Shields and Peyton Jones make use of overlapped instances, as well as two extensions to the type system: method constraints and closed classes. Unfortunately, neither of these extensions are implemented in Hugs or GHC today.

5.3.2 Haskell's overlooked object system

To date, *Haskell's overlooked object system* [13] is almost certainly the most comprehensive paper on object-oriented programming in Haskell. In contrast to the other papers reviewed in this thesis, the work described in this paper is not at all concerned with interoperability. Instead it focuses on applying object-oriented programming techniques directly (and exclusively) *in* Haskell. It goes beyond exposing an external object-oriented interface inside Haskell and instead implements the underlying object-oriented machinery for stateful objects, inheritance, and subtyping, in Haskell.

A large portion of the paper describes the OOHaskell library. By using this library (with Haskell 98, multi-parameter type classes, and functional dependencies) a Haskell programmer can apply object-oriented idioms to their Haskell programs.

The obvious question is: can we use OOHaskell to expose .NET concepts in Haskell? Each .NET class could be exposed in Haskell as an OOHaskell class. The methods and fields of these classes would, via the bridge, proxy those of the underlying .NET class. This sounds like a promising approach to the type and object system mapping problem, but there are a number of reasons why it is unsuitable:

Coverage Some important mapping problems are not addressed. OOHaskell implements core object-oriented concepts like object state, inheritance and subtyping, but it does not address some of the more difficult challenges associated with bridging to object-oriented systems. The method overloading and namespace problems identified in Chapter 4, are not addressed by OOHaskell. Since .NET uses both of these concepts extensively, this makes direct use of OOHaskell impracticable.

Redundancy Both OOHaskell and the .NET framework prescribe a certain object-oriented semantics. In the case of OOHaskell, the semantics is configurable and would have to precisely match .NET's semantics (a problematic task).

Complexity OOHaskell employs type-level programming in almost every aspect of its implementation. The resulting complexity of such extensive type-level programming is undesirable.

Heavyweight The OOHaskell library is not the sort of lightweight solution we are seeking for the bridge.

Even though the direct use of OOHaskell is not suitable for us, the paper itself is valuable. It comprehensively describes and assesses a range of techniques for encoding object-oriented class hierarchies in Haskell 98. Some variation or combination of these techniques, or the techniques used to create OOHaskell, may lead to an appropriate type and object system mapping for the bridge.

6

Salsa

During the course of the work on this thesis, a software bridge was developed that allows Haskell and .NET programs to interoperate. This bridge is called *Salsa*, and it serves two purposes:

- It satisfies the original goal of the thesis, stated in Chapter 2, of building a software bridge between Haskell and .NET.
- It provides a concrete realisation of the research results, in particular the work on overcoming the challenges involved in mapping the .NET type and object system into Haskell (as outlined in §4.2).

This chapter gives an overview of the design and implementation of Salsa, including how it addresses the challenges involved in runtime system interoperability, and type and object system mapping.

6.1 Design

A number of design decisions were made as part of building the bridge. The following statements describe the important aspects of the resulting design.

- The bridge must enable an existing Haskell runtime system (of a supported Haskell implementation) and the .NET execution engine to interoperate. This rules out any modification of the runtime systems to support the interoperation, and makes the bridge *lightweight*.
- Focus on providing convenient .NET access *from* Haskell, allowing access *to* Haskell from .NET only as necessary for using libraries. This restricts the scope of the bridge somewhat, leaving full bidirectional interoperability as future work.

- Implement the bridge core as a Haskell library, built on top of the Haskell C FFI. This makes the runtime system interoperation component of the bridge independent of any particular Haskell implementation.
- Have the Haskell executable host the .NET execution engine. Given the focus of accessing .NET *from* Haskell, hosting .NET in Haskell is the most appropriate hosting arrangement.
- Employ .NET reflection to automatically import .NET libraries for the bridge. This feature helps satisfy the goal of *convenience*.
- Support version 2.0 of the .NET framework.

Limitations of the design

The design points described in the previous section lead to some limitations for the bridge. They include:

Limited bidirectional interoperability Users who wish to embed Haskell in .NET applications, or require access to Haskell beyond what is required to use typical .NET libraries, will not find the bridge appropriate for their needs.

Efficiency Calling from Haskell into .NET, and from .NET into Haskell entails the unavoidable overhead of leaving one runtime and entering the other. Making inter-runtime calls in tight loops is not expected to give good performance.

6.2 Implementation

The Salsa implementation can be split into three main parts: a Haskell library, a .NET assembly, and a binding generator program.

6.2.1 Haskell library

As stated in the design section, Salsa provides .NET interoperability to Haskell programs through an ordinary Haskell library. This library provides the Haskell-side infrastructure for interacting with .NET, including functions for creating objects, invoking methods, and accessing properties. It also includes a type-level implementation of an overload resolution algorithm and implicit conversion algorithm to support convenient and type-safe object-oriented overloading and subtyping in the bridge.

The declarations in the modules of Salsa's Haskell library work together with the Haskell modules generated by the *binding generator* to provide access to specific .NET classes and their members from Haskell.

The library also contains code that loads the .NET execution engine into the process and allows .NET code in the *driver assembly* to be executed. This is implemented using the Haskell Foreign Function Interface (FFI) and a very thin wrapper for Microsoft's Component Object Model (COM).

6.2.2 .NET driver assembly

Salsa includes a .NET assembly called the *driver assembly*. This assembly provides the requisite support for the bridge on the .NET side, which includes the following functions:

Entry point Haskell programs using Salsa enter .NET initially via an entry point method in the driver assembly. This entry point provides access to function pointers for obtaining stub functions that can call arbitrary .NET code.

Stub generation Salsa provides arbitrary access to .NET by way of dynamically generated stub functions that can be called from Haskell via function pointers. The driver assembly includes .NET code that generates these stub functions on demand for specific operations, such as: invoking a particular constructor, or creating a .NET delegate instance that will call a Haskell function when invoked.

Object reference management The driver assembly maintains a collection of the .NET object instances that are currently being referred to from Haskell code. This collection is known as the 'in table'.

6.2.3 Binding generator

Salsa requires Haskell modules to be generated from the .NET metadata in order to provide access to the desired .NET libraries. A binding generator, written in C#, is provided for this purpose. The generator produces a number of Haskell modules containing type class instances, type function definitions, and foreign import declarations as appropriate for binding to the indicated .NET libraries.

Note: Salsa's Haskell library and .NET driver assembly are not generated. The generated Haskell modules are in addition to these core components.

6.3 Addressing the challenges

The following sections outline the techniques that are employed in Salsa to solve the challenges outlined in Chapter 4.

6.3.1 Runtime system interoperability

The challenges involved in getting the Haskell runtime system and .NET execution engine to interoperate, as outlined in §4.1, are broad rather than deep. Solving the problems is mostly a matter of careful coding, dealing with the .NET interoperability services and the Haskell FFI. The following points outline specifically how the challenges are addressed in Salsa.

Hosting Salsa uses the Haskell FFI to access the COM interface for the .NET execution engine (via a thin COM wrapper). This is used to load the execution engine into the Haskell process and call the entry-point method in the driver assembly.

Memory and reference management References to .NET objects are stored in Haskell as an Int32 value and a ForeignPtr. The integer is a key into the in-table maintained by the driver assembly. The in-table prevents .NET objects from being garbage collected prematurely, and allows objects to be marshaled by integer key (which avoids having to pin foreign-referenced .NET objects). The foreign pointer has a concurrent finaliser attached, which calls into .NET to inform the driver assembly when the object is no longer referenced from Haskell code.

Data marshaling The Haskell FFI and .NET marshaling systems are used by Salsa to handle most of the low-level data marshaling tasks. A small set of primitive types are marshaled by value using C types (which are supported by both systems). All other types, including other .NET value types, are marshaled by reference (i.e. as keys of the in-table).

Transfer of control The initial transfer of control from Haskell to .NET occurs through the driver assembly entry-point. All other inter-runtime transfers of control are triggered by either: calling a function pointer to a .NET delegate (for transferring from Haskell to .NET), or by calling a .NET delegate to a function pointer (for transferring from .NET to Haskell).

Threading Salsa uses a global lock to synchronise access to the in-table. The Haskell runtime that is used must support threading because of the .NET garbage collector, which calls finalisers on a different thread.

Since the interesting results pertain to the type and object system mapping, rather than runtime system interoperability, the remainder of this chapter — and this thesis — concentrates on the mapping aspects of Salsa.

6.3.2 Type and object system mapping

Subtyping The challenge of providing .NET-style subtyping in Haskell is addressed by:

- Applying type-function based implicit conversions on property assignments, method arguments and constructor arguments; combined with a type-class based implementation of coercion semantics for subtyping. Refer to §8.4 for details.
- Duplicating the bindings for inherited members in descendant classes.

Neither nested phantom types (used in Lambada [18], see §5.2.2) or type class hierarchies (used in MochΛ [20], see §5.2.1) are used to handle subtyping in Salsa. Chapter 8 contains a comprehensive explanation of Salsa's approach.

Method overloading The challenge of providing access to .NET-style overloaded methods in Haskell is addressed by:

- Incorporating a type-level overload resolution algorithm into the bindings to resolve overloaded method invocations at compile time, and to perform any required implicit conversions.
- Using a type class for dispatching method invocations to the appropriate implementation, combined with an associated type synonym to fix the method result type. A version of this technique is used in MochΛ, except using functional dependencies instead of associated type synonyms.
- Using tuple types to represent method arguments and method signatures (as in MochΛ).

Chapter 8 contains a comprehensive explanation of this new approach to object-oriented method overloading. It also covers the related approach to subtyping that is used in Salsa.

Namespaces The challenge of appropriately exposing .NET namespaces in Haskell is addressed on a number of fronts:

- The implicit namespace that .NET wraps around every type declaration is addressed by using first-class labels (like those used in HList [14] and OOHaskell [13]) for naming .NET members. By using labels, a single name can be given many meanings depending on the context in which it is used.
- .NET's explicit namespaces are handled by exporting all of the Haskell proxy types, that represent the .NET types of a particular namespace, into a particular Haskell module. The module is then given a hierarchical name that corresponds to the .NET namespace whose types it contains.
- Nested types, which are rarely seen in the interfaces of .NET libraries, are handled using a simple name-mangling scheme. The scheme is similar to .NET's internal naming system for nested types.

7

Mapping Types and Members with Labels

There are a number of .NET features that the bridge must to expose to the Haskell programmer in order to provide interoperability. .NET types, methods, properties; all of these need to have some representation in Haskell, and this representation should allow the features to be used in much the same way as a programmer of a typical .NET language would. Ideally they should look similar (syntax), behave similarly (semantics), and be — as much as possible — type-safe. All these attributes are important for Salsa given the goals of usability and safety.

This chapter describes the basic Haskell infrastructure that Salsa provides, and the patterns it follows, for exposing .NET types and their members in Haskell. The ubiquitous use of *labels* to do this is a worthwhile contribution, and leads to a very natural syntax for using .NET in Haskell.

7.1 Labels

The concept of a label, being a named language construct that represents itself and nothing more, is a designed-in feature of many languages. Lisp's *atomic symbols* [17, §1.1] are an early example of this feature. In Haskell, labels typically take the form of a singleton type:

```
data FatFree = FatFree
```

This declares a data type FatFree and a single constructor **FatFree** (which is the only non-bottom value for the type); and together these make a label. This idea is not new; it is seen frequently in the implementations of record systems for Haskell. Kiselyov, Lämmel and Schupke's work on HList [14] is a notable example.

Salsa's ubiquitous use of labels, for the purpose of exposing foreign entities in Haskell, is unlike previous Haskell bridges and bindings. Labels are employed not only for representing .NET types but also for *naming* the *members* of these types. The benefits of using labels in this way include:

Natural syntax Since labels consist of identically named type and data constructors, they can be used both on the value-level and on the type-level. This is useful for providing a natural syntax to the bridge, for example: the .NET type Button can be represented on the type-level with the type Obj Button and its .NET instance constructor can be named on the value-level with the data constructor **Button**.

Context sensitivity Haskell's type classes allow a single label to be interpreted in different ways by making the label type an instance of different type classes. This allows names to be overloaded in a convenient fashion.

Type-level programming Algorithms implemented at the type-level, such as those implemented with type functions (see §3.3), manipulate types instead of values. Labels can thus be directly processed by type-level programs.

7.2 Mapping types

The section outlines how Salsa represents .NET types in Haskell.

7.2.1 Primitive types

For the bridge to be useful, some .NET types must be represented in Haskell as *standard* Haskell types. Without this, all data entering Haskell from .NET would be opaque and Haskell would be unable to interpret it. In Salsa, the types that are represented as standard Haskell types are called *primitive types*.

Table 7.1 lists the .NET types that are considered to be primitive in Salsa, along with the associated Haskell type. Figure 7.1 defines the set 'prim' that contains the Haskell type representations of the primitive types.

Primitive types are always marshaled by value. Salsa marshals values of these .NET types to the corresponding Haskell types when accepting data from .NET, and vice versa when passing data to .NET.

Note: for usability Salsa treat strings as primitive types and marshals them by value, even though System.String is actually a reference type in .NET. This is safe because .NET strings are immutable.

.NET type	Haskell type
System.Int32	Int32
System.String	String
System.Boolean	Bool
System.Double	Double

Table 7.1: Mapping of primitive types

$$\frac{}{\mathsf{Int32}\ \text{prim}} \qquad \frac{}{\mathsf{String}\ \text{prim}} \qquad \frac{}{\mathsf{Bool}\ \text{prim}}$$

$$\frac{}{\mathsf{Double}\ \text{prim}}$$

Figure 7.1: Salsa primitive types

7.2.2 Reference types

Any .NET type that is not considered by the bridge to be a primitive type is treated as a *reference type*. This includes .NET classes, interfaces, and even value types (which are boxed into .NET references as necessary). Reference types are opaque to the Haskell program and can be directly manipulated only by .NET. They are marshaled as the integer keys of the bridge driver's in-table.

Reference types are represented in Haskell with the type Obj a, which has the following definition:

```
data Obj a = Obj ObjectId (ForeignPtr ())
           | ObjNull
```

Salsa declares a label for every .NET type that is bound. This label's type is applied to the phantom type parameter *a* to obtain the Haskell type representation of a .NET type. For example, the .NET type Object is represented in Haskell as the type Obj Object. This technique is commonly used in Haskell bridges [5, 18, 21], and it ensures that foreign references cannot be accidentally substituted for one another, even though all such references have the same representation at the value level.

There are two constructors for Obj values: **Obj** and **ObjNull**. **Obj** stores the integer key that corresponds to the real .NET object in the bridge driver's in-table. It also stores a ForeignPtr, to which a finaliser (which removes the object from the in-table when called) is attached. All non-null .NET object references use the **Obj** constructor; null references, which do not require finalisation, are

$$\frac{\tau \text{ class/interface label}}{\mathsf{Obj}\ \tau \text{ ref}}$$

Figure 7.2: Salsa reference types

instead represented with **ObjNull**.

As an optimisation, Salsa actually uses strict fields in the **Obj** constructor.

Figure 7.2 defines the set 'ref' containing the Haskell type representations of the bridge reference types.

7.3 Mapping members

The section outlines how Salsa represents the members of .NET types in Haskell, including: methods, constructors, properties, fields and events.

7.3.1 Methods

Salsa provides support for invoking the static and instance methods of .NET types, and it makes use of labels in order to do this. A label is declared for every unique method name that is made accessible from Haskell, and the data constructor for the label is directly employed in the syntax for invoking a method.

Invoking a method can be performed in Salsa using the **invoke** function shown in Listing 7.1; it takes three arguments:

1. The *target* of the method invocation. For instance methods, this argument must be a value of the appropriate object type (`Obj a`), and the value identifies the particular object on which the method will be invoked. For static methods, the argument must be the constructor for the label identifying the class in which the method to invoke is contained.

   In both cases, the *type* of the target is used to identify which class the method to be invoked is a member of.

2. The name of the *method* to invoke. The argument must be the constructor for the label associated with the method to be invoked.

3. A tuple of *arguments* for the method invocation.

```
invoke t m args = rawInvoke t m args

class Invoker t m args where
   type Result t m args
   rawInvoke :: t → m → args → Result t m args
```

Listing 7.1: Method invocation support

For example, to invoke the instance method ToString of an object, one can write:

```
invoke object1 ToString ()
```

And to invoke the static method WriteLine of the Console class, one can write:

```
invoke Console WriteLine ("Salsa")
```

The technique, of using labels to identify methods, is similar to part of the MochΛ implementation which uses this idea when dealing with overloaded methods. However MochΛ (and other Haskell bindings and bridges), ultimately provide a unique Haskell function for invoking each bound method. This differs from Salsa's approach, where method labels are *exposed* as part of the syntax for invoking methods (Salsa's **invoke** function is called directly by the Haskell programmer).

By directly exposing labels in this way, and not defining a Haskell function for each method in each .NET type, Salsa is able to use method *names* in a *context sensitive* manner. For example, consider the following .NET classes:

```
public class TennisPlayer
{
   public void Serve() {···}   // (A)
}

public class Waiter
{
   public void Serve() {···}   // (B)
}
```

Given a variable t of type Obj TennisPlayer and w of type Obj Waiter, the Serve method can be called like so:

```
invoke t Serve ()       -- invokes (A)
invoke w Serve ()       -- invokes (B)
```

Notice that the same label, Serve, refers to (A) in the context of a TennisPlayer object, and (B) in the context of a Waiter object, just like a method name in a C# program would. The **Invoker** type class, shown in Listing 7.1, is used to determine which method is intended by the programmer.

Constraining the result type

In addition to selecting the intended target method, **Invoker** constrains the result type of method invocations in order to avoid superfluous type annotations.

Pang and Chakravarty [21] describe a technique for constraining the result type of object-oriented method invocations using functional dependencies [9]. Salsa employs the same technique but using associated type synonyms [2] instead. As Listing 7.1 shows, the associated type synonym Result in **Invoker** directly constrains the result type of the **rawInvoke** function.

It is interesting to note that, in .NET, the result type of a method depends not only on the target type and the method name, but also on the argument types. This is the case because it is possible to have two methods of the same name, invocable through the same class, that have different result types[1]. The situation arises when a method is overloaded, *and* an overload with a different signature and result type is defined in the parent type. For example:

```
public class C
{
  public Int32 M(Boolean b) {···}   // (A)
}

public class D : C
{
  public String M(Double d) {···}   // (B)
}
```

The method M can be invoked on an object of type D in two ways:

- with a boolean argument, yielding an Int32 value, or
- with a double argument, yielding a String value.

[1]There does not need to be any relationship (subtype or otherwise) between the two result types.

$x \mathbin{\#} f = f\ x$

Listing 7.2: Reverse application operator

This possibility is handled cleanly in Salsa by including the argument signature tuple as one of the type arguments to the associated type synonym, Result.

Note: The actual implementation of **invoke** and **Invoker** in Salsa is more complicated than that of Listing 7.1 due to Salsa's support for overload resolution and implicit conversion (which is described in Chapter 8).

Improving the syntax

There is a common trick for improving the syntax of object-oriented method invocations in Haskell. This trick involves using an operator that performs function application with the arguments flipped. It was first described by Peyton Jones, Meijer and Leijen [22] in their work on binding COM components in Haskell.

They define #, the reverse function application operator, as shown in Listing 7.2, and define *f* so that it accepts the target of the invocation as its last argument. Unfortunately, this technique is incompatible with our direct use of labels. Our *f* is always a label constructor, not a function that can be used to directly invoke a method, and there is little that can be done to resolve this because function application has the highest precedence in Haskell. The expression:

button1 # *Click* ()

is always interpreted as:

button1 # (*Click* ())

which fails since *Click* is a nullary data constructor.

In order to support this very convenient syntax, Salsa defines a function for every label. This function calls **invoke** partially applied with the label's data constructor. In the case of the Click label, Salsa defines the function **_Click**:

_Click args t = *invoke t Click args*

This allows the original syntax to be used, with the caveat that the method name be prefixed with an underscore (giving the function rather than the data constructor):

button1 # *_Click* ()

Note that despite the use of a function to identify the method to invoke, this technique retains the original benefits of using labels. The **_Click** function is only associated with the Click label, and not the Click member of the Button class (or any other class).

7.3.2 Constructors

Salsa supports the invocation of instance constructors to create instances of .NET objects. Internally, instance constructors are treated as static methods with the label Ctor[2]:

data *Ctor* = *Ctor*

Externally however, Salsa provides a natural, C#-like syntax. This syntax takes the form of a **new** function, which is simply a wrapper around the **invoke** function partially applied with the Ctor method label:

new t args = *invoke t Ctor args*

As with all static methods bound by Salsa, the argument for the target of the invocation must be the constructor for a label; where the label is associated with the type containing the member to be invoked. In the case of an instance constructor, this is the type being instantiated. The resulting syntax is very similar to that of C#:

button1 ← *new Button* ()

Note: Static (or class) constructors are not bound by Salsa. Such constructors cannot be invoked by any .NET language (the runtime invokes them automatically).

7.3.3 Fields, properties and events

Support for reading and writing fields, getting and setting properties, and adding and removing events, is provided in Salsa through its attribute system. This system provides a convenient syntax for reading, writing and updating such values. It was inspired by the attribute systems of other Haskell bindings including Gtk2Hs [29], wxHaskell [15] and hs-fltk [11].

[2]This is analogous to .NET's internal treatment of instance constructors as static methods with the name '.ctor'.

The Salsa system differs from the other systems in a number of ways, including:

- Use of labels to identify attribute names. Just as Salsa uses labels to identify method names, Salsa uses labels to identify the attribute names corresponding to .NET fields, properties and events. This differs from the other attribute systems which use functions instead of labels for this purpose.
- Support for adding and removing event listeners. Earlier attribute systems provide only read/write style access, which does not fit the model of .NET events where adding and removing events both behave like write operations.
- Support for implicit conversions when assigning to properties and fields. Refer to §8.4 for more information of Salsa's implicit conversion implementation.

As with Salsa's method implementation, the use of labels in the attribute system allows the names of .NET fields, properties and events to overlap with other names that are bound by Salsa from the .NET libraries.

8

Overloading and Subtyping with Type Functions

8.1 Introduction

This chapter presents a novel solution to the problem of encoding object-oriented overloading and subtyping in Haskell (extended with type families [1, 26]). The technique is employed in Salsa to expose .NET libraries — that make extensive use of both method overloading and subtyping — in Haskell. In addition, access to the libraries is type-safe and does not require superfluous type annotations.

Previous work (see Chapter 5) offers solutions to this problem that are type-safe, and avoid superfluous type annotations, but do not allow the simultaneous use of method overloading and subtyping [20, 21]. Methods whose signatures differ on types where one is a subtype of the other, cannot be encoded.

Unfortunately, some commonly used methods in the .NET framework are overloaded in this way. Take the Write method from the Console class, for example:

```
public static class Console
{
  public void Write(String s)  {···}   // (A)
  // ...
  public void Write(Object o) {···}   // (B)
}
```

Here String is a subtype of Object. If Write is called with an argument of type String, which method should be called: (A) or (B)? A String is an acceptable argument for both methods.

The C# overload resolution algorithm, which is executed at compile-time by the C# compiler to resolve such method invocations, gives us the answer: the String overload is chosen because the conversion String → String is *better than* String → Object (where *better than* is a relation defined in the C# specification).

Unsurprisingly, while C# compilers implement the C# overload resolution algorithm, Haskell compilers do not. However, we know (from §3.3) that arbitrary computations can be expressed at the type-level using type functions (in Haskell compilers that support them). Can this be exploited so that the Haskell compiler resolves overloaded method invocations as it type checks the program? The following sections show that this is not only possible, but also that the resulting solution is rather elegant.

In addition to resolving overloading at compile time, a related technique is described to perform implicit conversions (based on the subtype relation). This is used in Salsa to support subtyping where overload resolution does not apply, such as when assigning a value (of type τ_1) to a variable (of type τ_2 where $\tau_1 <: \tau_2$).

8.2 Type equality

Before we can implement a type-level algorithm that deals with .NET types, we need a way of comparing .NET types for equality. More precisely, we need a type-level binary predicate function **TyEq** that returns a type-level boolean indicating if the given .NET types are equal or not:

$$\textbf{type family}\ \mathit{TyEq}\ \tau_1\ \tau_2$$

According to Shields [27], C# is nominally typed with the exception of (the structurally typed) arrays and unsafe pointer types. For the purposes of the bridge we define two types to be equal if they have the same assembly-qualified name.[1]

Since the Salsa bindings provide a unique Haskell label (or a unique Haskell type in the case of a primitive type) for every .NET type encoded by the bridge, we can define *.NET* type equality, in the Haskell type-system, as a relation over *Haskell* types.

The *equality constraints* feature, introduced with type families, allows us to express the requirement that two types be the same in the context of a function, class or instance declaration. This sounds like a promising way of defining the type equality predicate, but unfortunately it is not enough. We require a type function that returns true if the types are equal, and false otherwise; equality

[1]A type's assembly-qualified name includes the full name of the type, including any namespace, in addition to the name and version of the assembly in which it is defined.

constraints cannot give us this. For example, take the following definition of **TyEq**:

$$
\begin{array}{llll}
\textbf{type instance } \mathit{TyEq}\ (\tau_1 \sim \tau_2) \Rightarrow & \tau_1\ \tau_2 = \mathit{TTrue} & & \text{-- WRONG} \\
\textbf{type instance } \mathit{TyEq} & \tau_1\ \tau_2 = \mathit{TFalse} & & \text{-- WRONG}
\end{array}
$$

This definition is not permitted because the heads of the two instance declarations are overlapping ($\tau_1 \equiv \tau_1 \wedge \tau_2 \equiv \tau_2$), while the bodies are distinct ($\mathit{TTrue} \not\equiv \mathit{TFalse}$). Type instance definitions are not allowed to overlap unless their bodies are syntactically identical.

For the same reason, the following (longer) definition is also not a solution:

$$
\begin{array}{llll}
\textbf{type instance } \mathit{TyEq}\ \mathit{String} & \mathit{String} & = \mathit{TTrue} & \\
\textbf{type instance } \mathit{TyEq}\ \mathit{Int32} & \mathit{Int32} & = \mathit{TTrue} & \\
\ldots & & & \\
\textbf{type instance } \mathit{TyEq}\ \tau_1 & \tau_2 & = \mathit{TFalse} & \text{-- WRONG}
\end{array}
$$

The only remaining option for a direct implementation of **TyEq** requires an instance declaration for every possible pair of types over which the type equality relation is defined. Each such definition would indicate if the types are equal (*TTrue*) or not (*TFalse*). Given the size of the .NET framework libraries, a type function implementation that requires $\mathcal{O}(n^2)$ instance declarations is simply not feasible for the bridge. (Even with a code generator producing the instances, it is simply too much data to expect a Haskell compiler to process.)

A solution to this dilemma, and the solution implemented in Salsa, is to associate a Gödel number with each type and then define **TyEq** using equality over the Gödel numbers. This reduces the number of instance declarations from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$.

We represent Gödel numbers as type-level lists of booleans[2]. Given the definitions for type-level lists and booleans from §3.3, Listing 8.1 defines type-level equality for booleans (**BoolEq**) and lists (**ListEq**).

The Gödel numbering scheme is defined by the type function **TyCode**, shown in Listing 8.2, which returns a unique type-level list of booleans for the given type. Implementing **TyEq** is then just a matter of comparing the lists returned by **TyCode** for the given types, as shown in Listing 8.3. The listing also defines

[2]Peano numerals could have been used instead of boolean lists. However, definitions for booleans and lists are required elsewhere in the implementation so reusing them here makes sense. The lists also require less type-level evaluation by the compiler, since they have a nesting depth of $\mathcal{O}(\log n)$ rather than $\mathcal{O}(n)$.

```
type family   BoolEq x      y
type instance BoolEq TTrue  TTrue  = TTrue
type instance BoolEq TFalse TFalse = TTrue
type instance BoolEq TFalse TTrue  = TFalse
type instance BoolEq TTrue  TFalse = TFalse

type family   ListEq xs         ys
type instance ListEq TNil       TNil       = TTrue
type instance ListEq (x ::: xs) TNil       = TFalse
type instance ListEq TNil       (x ::: xs) = TFalse
type instance ListEq (x ::: xs) (y ::: ys) = TAnd (BoolEq x y) (ListEq xs ys)
```

Listing 8.1: Type-level equality for booleans and lists

```
type family   TyCode τ
type instance TyCode Int32        =          TFalse ::: TNil
type instance TyCode String       =          TTrue  ::: TNil
type instance TyCode (Obj Object) = TTrue ::: TFalse ::: TNil
...
```

Listing 8.2: Gödel numbering for types

```
type family   TyEq τ1 τ2
type instance TyEq τ1 τ2 = ListEq (TyCode τ1) (TyCode τ2)

type family   TyListEq m          n
type instance TyListEq TNil       TNil       = TTrue
type instance TyListEq TNil       (n ::: ns) = TFalse
type instance TyListEq (m ::: ms) TNil       = TFalse
type instance TyListEq (m ::: ms) (n ::: ns) = TAnd (TyEq m n)
                                                     (TyListEq ms ns)
```

Listing 8.3: Type-level type equality and type list equality

$$\frac{\tau_1 = \tau_2}{\tau_1 <: \tau_2} \qquad \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$\frac{\textbf{class } C : C' + \overline{I}\ \{\cdots\}}{C <: C'} \ (\text{C-B}{\small\text{ASE}}^{a})$$

$$\frac{\textbf{class } C : C' + \overline{I}\ \{\cdots\}}{C <: I_i} \ (\text{C-I}{\small\text{MP}})$$

$$\frac{\textbf{interface } I : \overline{I'}\ \{\cdots\}}{I <: I'_i}$$

[a] If a class has no base class, C' is object.

Figure 8.1: Subtyping rules for C# (based on [27])

TyListEq which compares lists of types for equality. This function is useful for dealing with method signatures represented as a list of the types of the method arguments.

8.3 Encoding subtyping

In addition to comparing types for equality, our type-level algorithm must be able to determine if a particular .NET type is a subtype of another. In this section we develop a type function, **IsSubtypeOf**, that implements this predicate.

Subtyping is present in .NET (and in the C# language) through two related mechanisms:

Single inheritance of classes. Every class (with the exception of the base Object type) *derives from* exactly one class. If an explicit base class is not provided, Object is assumed.

Multiple inheritance of interfaces. Each class/interface *implements* zero or more interfaces.

The reflexive, transitive closure of the *derives from* and *implements* relations form the subtype relation (<:) defined in Figure 8.1 (these subtyping rules are based on work by Shields [27]). We define this relation for every .NET type, including .NET value types which, with the exception of primitive types, are always boxed as reference types when passed through the bridge.

We implement **IsSubtypeOf** in terms of another type function, **SupertypesOf**, which returns a list of the supertypes of a particular class as defined by the subtype relation. Formally:

$$\textbf{SupertypesOf}\ \tau = \{\tau' \mid \tau <: \tau'\}$$

Listing 8.4 shows the implementation of **IsSubtypeOf**, **SupertypesOf** and the helper function **TyElem** (which evaluates to true if and only if the given type is in the given list).

In Salsa, the list of supertypes for a class is calculated by the code generator and inserted into the source (as a constant type-level list in the appropriate **SupertypesOf** type instance) when the bindings for the class are generated. An alternative would be to calculate the list of supertypes during type checking based on more primitive relations (such as *derives from* and *implements*).

The implementation of **IsSubtypeOf** is reasonably efficient because the list of supertypes that it searches is typically short.

```
type family   SupertypesOf τ
type instance SupertypesOf (Obj Object) = TNil
type instance SupertypesOf (Obj Button) = (Obj Object) ::: TNil

type family   TyElem τ1 ts
type instance TyElem τ1 TNil      = TFalse
type instance TyElem τ1 (t ::: ts) = TOr (TyEq τ1 t) (TyElem τ1 ts)

type family   IsSubtypeOf τ1 τ2
type instance IsSubtypeOf τ1 τ2 = TOr (TyEq τ1 τ2)
                                      (TyElem τ2 (SupertypesOf τ1))
```

Listing 8.4: Type-level subtype predicate

8.4 Implicit conversions

When invoking a method or assigning a value to a variable (and in some other situations [4, §13.1]), the C# compiler will automatically apply *implicit conversions* to convert a value of one type to another (compatible) type. Assigning a value of type Int32 to a field of type Double, for example, will cause the compiler to automatically convert the Int32 value to a Double value.

If an implicit conversion exists from one type to another, the required conversion (if any) will always succeed at run-time. This is why the compiler can insert them automatically (and hence why they have the name 'implicit').

Without a system that performs these implicit conversions where necessary, the source code is polluted with seemingly superfluous type casts and is, consequently, less readable. In case of the bridge, this must be avoided to satisfy the goal of usability.

This section defines an appropriate implicit conversion relation for the bridge, implements it as a type function, and then demonstrates how the type function is employed to perform an implicit conversion for a hypothetical Haskell function that assigns a value to a .NET field.

8.4.1 Definition

The C# specification [4] defines the set of valid implicit conversions in §13.1, however the set is inappropriate for the bridge for a number of reasons, including:

- The rules do not deal with the primitive bridge types (described in §7.2.1). We need additional rules to ensure that the primitive bridge types behave like the corresponding .NET types would in C#.
- There are many rules in the C# specification that do not apply in the context of the bridge and can be removed for simplicity. These rules relate to value types (which are never seen by the Haskell side of the bridge) and features that the bridge does not support (like user-defined conversions).

Figure 8.2 gives a set of inference rules for the implicit conversion relation that is employed by Salsa. These rules define the relation $\tau_1 \leadsto \tau_2$ which holds if and only if a type τ_1 is implicitly convertible to τ_2. The types involved are Haskell types (which include the proxy types representing .NET types in Haskell). The relation must be defined on Haskell types because the conversions are applied by the Haskell compiler and need to include the primitive bridge types like Int32.

The rules cover the following types of conversions:

Identity A value of a type can, trivially, be converted to the same type (C-Id). This corresponds to C#'s identity conversion.

Subtyping If τ_1 is a subtype of τ_2, we allow converting from τ_1 to τ_2 (C-Subtype). This corresponds to C#'s implicit reference conversions.

Numeric conversions Only one of C#'s implicit numeric conversions, Int32 $\leadsto$ Double, applies to the types exposed by the bridge (C-Num1).

Null literal The *null* literal has a special type which can be implicitly converted to any reference type (C-Null).

Boxing We automatically *box* primitive types into a .NET `Object` instance (C-Box) if needed. This is similar to C#'s implicit boxing conversions for value types.

8.4.2 Implementation

The type function **ConvertsTo**, shown in Listing 8.5, defines the implicit conversion predicate ($\rightsquigarrow$). It returns a type-level boolean indicating if the first type can be implicitly converted to the second.

The function is implemented as a sequence of boolean disjunctions, one for each of the five inference rules from Figure 8.2. It makes use of many of the type functions defined earlier, along with **IsPrim** and **IsRef**, which indicate if a given type is a member of, respectively, the 'prim' or 'ref' sets defined in §7.2. Listing 8.6 shows the implementation of these functions.

```
type family   ConvertsTo τ1 τ2
type instance ConvertsTo τ1 τ2 =
  (TOr (TyEq τ1 τ2)                                         (C-Id)
  (TOr (TAnd (IsPrim τ1)          (TyEq τ2 (Obj Object)))  (C-Box)
  (TOr (TAnd (TyEq τ1 Int32)      (TyEq τ2 Double))        (C-Num1)
  (TOr (TAnd (TyEq τ1 (Obj Null)) (IsRef τ2))              (C-Null)
       (TAnd (TAnd (IsRef τ1) (IsRef τ2))                  (C-Subtype)
             (IsSubtypeOf τ1 τ2))))))
```

Listing 8.5: Type-level implicit conversion predicate

```
type family   IsPrim τ
type instance IsPrim Int32   = TTrue
type instance IsPrim String  = TTrue
type instance IsPrim Bool    = TTrue
type instance IsPrim Double  = TTrue
type instance IsPrim (Obj τ) = TFalse

type family   IsRef τ
type instance IsRef τ = TNot (IsPrim τ)
```

Listing 8.6: Type-level 'prim' and 'ref' predicates

We now have all the required machinery to determine, at compile time, if a particular type can be implicitly converted into another type. This is great, but we need more. In addition to being able to determine *if* an implicit conversion is allowed, we must actually *perform* these conversions where they are required.

Since .NET employs *subset semantics* for reference types, a conversion from one reference type (Obj a) to another (Obj b) does not require a conversion at run-time. We do however need to inform the Haskell type system that the value has been subsumed to a different type. We use the Haskell function **unsafeCoerce** to do this.

The use of **unsafeCoerce** in this context is perfectly safe because the type parameter of Obj is a *phantom type* parameter. The type is not used in the definition of any of Obj's constructors, and thus they all have the same representation and can be safely coerced.

For value types, .NET employs *coercion semantics* for subtyping. Since some of the bridge's primitive types represent .NET value types (Int32, Bool and Double), the bridge must explicitly convert between these values.

The following list describes the types of conversions that can arise, and the required semantics for each:

Reference Obj a $\rightsquigarrow$ Obj b is coerced using **unsafeCoerce**.

Numeric widening Int32 $\rightsquigarrow$ Double is coerced with **fromIntegral**.

Identity $\tau \rightsquigarrow \tau$ does not require any coercion to be performed.

Boxing $\tau \rightsquigarrow$ Obj τ (where $\tau \in$ prim) is coerced with a boxing conversion function. This requires a call into .NET to obtain a reference to a boxed copy of the value.

We use a type class **Coercible** to encapsulate these conversions, and allow dispatching to the appropriate conversion operation based on the types of the values involved. Listing 8.7 shows the implementation of **Coercible**.

In contrast to **ConvertsTo**, which defines what conversions are allowed to be performed implicitly, the class **Coercible** operates at a lower level; it defines the conversions that are possible (but not necessarily implicitly performed) between types of possibly different representations. For example, **coerce** will perform the nonsense conversion from a value of type Obj Button to a value of type Obj Console, even though there is no meaningful relationship between the two class types.

$$\frac{}{\tau \leadsto \tau} \quad \text{(C-ID)}$$

$$\frac{\tau_1 \text{ ref} \qquad \tau_2 \text{ ref} \qquad \tau_1 <: \tau_2}{\tau_1 \leadsto \tau_2} \quad \text{(C-SUBTYPE)}$$

$$\frac{\mathsf{Obj\ Null} \text{ type of the null literal} \qquad \tau \text{ ref}}{\mathsf{Obj\ Null} \leadsto \tau} \quad \text{(C-NULL)}$$

$$\frac{\tau \text{ prim}}{\tau \leadsto \mathsf{Obj\ Object}} \quad \text{(C-BOX)} \qquad \frac{}{\mathsf{Int32} \leadsto \mathsf{Double}} \quad \text{(C-NUM1)}$$

Figure 8.2: Rules for implicit conversions

```
class Coercible from to where
  coerce :: from → to

instance Coercible Int32   Int32   where coerce = id
instance Coercible String  String  where coerce = id
instance Coercible Bool    Bool    where coerce = id
instance Coercible Double  Double  where coerce = id
instance Coercible Int32   Double  where coerce = fromIntegral
instance Coercible (Obj f) (Obj t) where coerce = unsafeCoerce
...
```

Listing 8.7: Coercible class to coerce values

8.4.3 Application

As an example of supporting implicit conversions using the implementation described in the previous section, we bind the following static .NET field in Haskell.

public static Double x;

We would like to be able to assign values to this field using a Haskell function **setX** that accepts a new value for the field and returns an IO action that performs the assignment. Something like this:

$$\begin{array}{l} \mathit{setX} :: \mathit{Double} \rightarrow \mathit{IO}\ () \\ \mathit{setX} = \cdots \end{array}$$

Except that the above implementation can only be supplied with values that are exactly of the type Double. In order to accept values of any type that can be implicitly converted to Double, we implement **setX** like so, employing the **ConvertTo** type function and **Coercible** type class:

$$\begin{array}{l} \mathit{setX} :: (\mathit{ConvertsTo}\ v\ \mathit{Double}\ \sim\ \mathit{TTrue}, \\ \qquad\qquad \mathit{Coercible}\ v\ \mathit{Double}) \Rightarrow \\ \qquad\qquad v \rightarrow \mathit{IO}\ () \\ \mathit{setX}\ v = \mathit{rawSetX}\ (\mathit{coerce}\ v) \\ \\ \mathit{rawSetX} :: \mathit{Double} \rightarrow \mathit{IO}\ () \\ \mathit{rawSetX} = \cdots \end{array}$$

Here **setX** will accept any type that can be implicitly converted to a Double. It uses **coerce** to perform the required coercion (if any) on the actual value that is passed in, before passing the value (which is now a Double) to **rawSetX**. **rawSetX** then performs the low-level task of assigning the value to the .NET field.

Of particular importance is **setX**'s type signature; it states that the argument type v must be:

1. implicitly convertible to a Double according to ($\rightsquigarrow$):
   $\mathit{ConvertsTo}\ v\ \mathit{Double}\ \sim\ \mathit{TTrue}$
   as well as,
2. (explicitly) coercible to a Double:
   $\mathit{Coercible}\ v\ \mathit{Double}$

The first constraint can be viewed as the entry point to the type-level implicit-conversion checking algorithm. It 'calls' the type function **ConvertsTo** with the appropriate arguments, and then uses an equality constraint to ensure that it evaluates to true.

The second constraint simply ensures that the required **Coercible** instance exists so that **coerce** can be called. (The required instance will exist if the first constraint is satisfied.)

Salsa uses the above technique in its property system in order to support assigning values to .NET properties and fields. It also plays a role in Salsa's support for invoking overloaded methods and constructors (which perform implicit conversions on their arguments if required).

8.5 Overload resolution

When compiling a method or a constructor invocation (and in some other situations [4, §14.4.2]), the C# compiler often has a choice to make. For example, .NET allows a class to provide several implementations of a particular method, all with the same name, and the compiler has to (statically) select which particular implementation should be invoked.

A set of method implementations sharing a name is called a *method group*. If there is more than one method in a group, the method is said to be *overloaded*. .NET requires that the signatures of the methods in a method group differ (where a signature is roughly the list of argument types), therefore a method name and signature is enough to uniquely identify a particular method in a class.

When a method is invoked, the compiler's *overload resolution algorithm* is used to choose the appropriate implementation to invoke, based on:

- the static type of the class on which the method was invoked;
- the name of the method (group); and
- a list of the static types of the arguments that have been supplied to the method invocation.

Like the C# compiler, the bridge has to perform overload resolution for method and constructor invocations. In the following sections we define an appropriate overload algorithm for the bridge, and then implement it such that the Haskell compiler performs the resolution when it type checks the program.

8.5.1 Definition

We base the overload resolution algorithm in Salsa on the C# overload resolution algorithm which is defined in §14.4.2 of the C# specification [4]. Consider a method invocation:

$$o.M(a_1, a_2, ..., a_n);$$

The algorithm would perform the following tasks in order to resolve it:

1. Enumerate the list of candidate methods for the invocation. This includes all methods named M in the (static) class of the object instance o, as well as any such methods in any superclass, unless a superclass method is *hidden* by a method in a derived class with the same signature.[3]

2. Filter the candidate methods, leaving only methods that are *applicable* given the argument list ($\overline{a}$). A method is applicable only if it has the same number of formal parameters as the invocation has arguments, *and* there is an implicit conversion from each argument type to the respective formal parameter type of the candidate method. (App)

3. Find the *best* method from the remaining candidate methods. Intuitively, a candidate method is better than another candidate method if the respective argument types are more 'easily' converted to the method's parameter types. More precisely, a method $\mathcal{M}$ is better than another method $\mathcal{N}$, in the context of a given argument list if (M-Better):

   - There is some argument where the *conversion* from the argument type to the corresponding parameter type in $\mathcal{M}$ is *better* than the conversion from the argument type to the respective parameter type in $\mathcal{N}$, *and*
   - No conversion from an argument type to the respective parameter type in $\mathcal{N}$ is better than the respective conversion for $\mathcal{M}$.

   Where a conversion from τ to τ_1 ($\mathcal{C}_1$) is said to be *better* than a conversion from τ to τ_2 ($\mathcal{C}_2$) if either of the following applies:

   - $\mathcal{C}_1$ is the identity conversion ($\tau \rightarrow \tau$) and $\mathcal{C}_2$ is not. (C-Better1)
   - There is a non-identity conversion from τ_1 to τ_2. (C-Better2)

[3]This behaviour is known as 'hide by name-and-signature' semantics. .NET also supports 'hide-by-name' semantics, where a single method hides all methods of the same name in the base class. C# uses 'hide by name-and-signature' semantics, as does Salsa.

$$\frac{|\overline{a}| = |\overline{m}| \qquad \forall a_i \rightsquigarrow m_i}{\overline{a} \;\vdash\; \overline{m}\ \text{app}} \quad (\textsc{App})$$

Figure 8.3: Rule for applicable candidate methods

$$\frac{\tau \neq \tau_2}{\tau \rightsquigarrow \tau \;>_c\; \tau \rightsquigarrow \tau_2} \quad (\textsc{C-Better1})$$

$$\frac{\tau_1 \rightsquigarrow \tau_2 \qquad \tau_2 \not\rightsquigarrow \tau_1}{\tau \rightsquigarrow \tau_1 \;>_c\; \tau \rightsquigarrow \tau_2} \quad (\textsc{C-Better2})$$

Figure 8.4: Rules for comparing conversions

4. If a single *best* method was found in the previous step, choose that method, otherwise indicate that a compile time error has occurred.

A similar sequence of steps are performed for constructor invocations (which are called using C#'s **new** operator).

Since step 1 can be performed prior to the compilation, it is performed by Salsa's code generator and the process is not covered further in this section.

The remaining steps (2–4) must be performed at compile time[4] because they depend on the types of the arguments supplied to the method invocation.

8.5.2 Value-level implementation

Having defined the overload resolution algorithm, we are now ready to implement it. Since the algorithm's implementation is non-trivial, we examine an equivalent value-level Haskell implementation before covering Salsa's type-function-based implementation. The value-level implementation is much clearer due to the use

[4]Confusingly, compile-time for the Haskell program is run-time for the resolution algorithm.

$$\frac{\begin{array}{c}\exists i.\ a_i \rightsquigarrow s_i \;>_c\; a_i \rightsquigarrow t_i \\ \forall i.\ a_i \rightsquigarrow t_i \;\not>_c\; a_i \rightsquigarrow s_i\end{array}}{\overline{a} \;\vdash\; \overline{s} \;>_m\; \overline{t}} \quad (\textsc{M-Better})$$

Figure 8.5: Rules for comparing methods under an applicable invocation

of higher-order functions (which have to be manually specialised in the type function implementation).

The overload resolution algorithm is implemented by the function **resolve** shown in Listing 8.8. This function takes:

- a list of candidate members (*ms*) corresponding to the methods of a particular method group,
- a list of argument types (*as*) from the call being resolved,

and returns the best applicable member from *ms* with respect to *as*. This is the member which should be invoked for the particular invocation. If there is no unique best applicable member, **resolve** fails.

```
resolve :: [Member] → [Type] → Member
resolve ms as = m
  where apps = filter (isApp as) ms
        [m]  = filter (isBestMember as) apps

        isApp as m = (length as ≡ length m) ∧
                     and (zipWith (⇝) as m)

        isBestMember as m = all (isBetterMember as m)
                                (filter (≢ m) apps)
```

Listing 8.8: Value-level overload resolution algorithm

The **resolve** function performs step 2 of the resolution algorithm by filtering the list of candidate methods on the function **isApp**. The function **isApp** implements the (App) rule from Figure 8.3, evaluating to true if and only if the given member is applicable with respect to the argument list.

Step 3 of the resolution algorithm is performed by filtering the list of *applicable* function members with **isBestMember** to select only the best members, i.e. methods that are better than all the other applicable candidate methods. An application of *isBestMember as m* evaluates to true if and only if *m* is better than all other members in *ms*, with respect to the argument types *as*.

Listing 8.9 shows the implementation of **isBetterMember** and a helper function **isBetterConv**. These functions correspond, respectively, to the $>_c$ and $>_m$ relations of Figure 8.4 and Figure 8.5.

$$
\begin{array}{l}
\mathit{isBetterConv} :: \mathit{Type} \rightarrow \mathit{Type} \rightarrow \mathit{Type} \rightarrow \mathit{Bool} \\
\mathit{isBetterConv}\ \tau\ \tau_1\ \tau_2 = (\tau\ \ \equiv \tau_1 \wedge \neg\,(\tau\ \ \equiv \tau_2)) \vee \\
\qquad\qquad\qquad\qquad\quad (\tau_1 \rightsquigarrow \tau_2 \wedge \neg\,(\tau_2 \rightsquigarrow \tau_1)) \\
\\
\mathit{isBetterMember} :: [\,\mathit{Type}\,] \rightarrow \mathit{Member} \rightarrow \mathit{Member} \rightarrow \mathit{Bool} \\
\mathit{isBetterMember}\ \mathit{as}\ m_1\ m_2 = \mathit{someBetter}\ s\ t\ \wedge \\
\qquad\qquad\qquad\qquad\qquad\quad \neg\,(\mathit{someBetter}\ m_2\ m_1) \\
\quad \textbf{where}\ \mathit{someBetter}\ m_1\ m_2 = \mathit{or}\ (\mathit{zipWith3}\ \mathit{isBetterConv}\ \mathit{as}\ m_1\ m_2)
\end{array}
$$

Listing 8.9: Value-level $>_c$ (better conversion) and $>_m$ (better member) predicates

8.5.3 Type-level implementation

This section presents a type-level implementation of the value-level overload resolution algorithm described previously. Listing 8.10 shows the type-level implementation of **resolve** (previously from Listing 8.8), and Listing 8.11 shows the implementations of the $>_c$ and $>_m$ predicates (previously from Listing 8.9).

Many of the type functions defined in earlier sections are used in the implementation, including the predicate for implicit conversions, **ConvertsTo**.

In the type-level implementation, the value-level Type and Member types, become real Haskell types and type-level lists of these types, respectively.

Tuples and lists

Type-level lists (of types) are convenient for implementing type-level algorithms because they allow the use of list recursion (which is used extensively in the overload resolution algorithm).

Isomorphic to type-level lists are tuple types, which also represent a sequences of types. Using tuple types for the bridge bindings is desirable because they provide a concise and natural looking syntax for constructs like method and constructor invocations.

In order to have the benefits of both type-level lists and tuples, we define two helper functions, **TupleToList** and **ListToTuple**, for converting between tuple types and type-level lists (of types). Listing 8.12 shows their definitions. Note that the definition for **TupleToList** requires the 1-tuples to be defined for every bridge type to prevent overlapping with the other instances.

Listing 8.13 shows an updated implementation of **Resolve** that accepts and

```
type family Resolve as ms
type instance Resolve as ms =
  FromSingleton (FilterBestMembers as (FilterApp as ms)
                                     (FilterApp as ms))

type family FromSingleton xs
type instance FromSingleton (x ::: TNil) = x

type family   FilterApp as ms
type instance FilterApp as TNil       = TNil
type instance FilterApp as (m ::: ms) =
  TIf (IsApp as m)  (m ::: FilterApp as ms)
                    (FilterApp as ms)

type family   IsApp as        ss
type instance IsApp TNil      TNil      = TTrue
type instance IsApp TNil      (s ::: ss) = TFalse  -- different lengths
type instance IsApp (a ::: as) TNil      = TFalse  -- different lengths
type instance IsApp (a ::: as) (s ::: ss) = TAnd (ConvertsTo a s)
                                               (IsApp as ss)

type family   FilterBestMembers as ms ns
type instance FilterBestMembers as ms TNil      = TNil
type instance FilterBestMembers as ms (n ::: ns) =
  TIf (IsBestMember as ms n) (n ::: (FilterBestMembers as ms ns))
                             (FilterBestMembers as ms ns)

type family IsBestMember as ms n
type instance IsBestMember as TNil       n = TTrue
type instance IsBestMember as (m ::: ms) n =
  TIf (TyListEq m n) (IsBestMember as ms n)
                     (TAnd     (IsBetterMember as n m)
                               (IsBestMember as ms n))
```

Listing 8.10: Type-level overload resolution algorithm

```
type family   IsBetterConv τ τ1 τ2
type instance IsBetterConv τ τ1 τ2 =
   TOr (TAnd (TyEq τ τ1) (TNot (TyEq τ τ2)))                    (C-BETTER1)
       (TAnd (ConvertsTo τ1 τ2) (TNot (ConvertsTo τ2 τ1))) (C-BETTER2)

type family   AnyBetterConv as         ss         ts
type instance AnyBetterConv TNil       TNil       TNil       = TFalse
type instance AnyBetterConv (a ::: as) (s ::: ss) (t ::: ts) =
   TOr (IsBetterConv a s t) (AnyBetterConv as ss ts)

type family   IsBetterMember as ss ts
type instance IsBetterMember as ss ts =
   TAnd (AnyBetterConv as ps qs) (HNot (AnyBetterConv as qs ps))
```

Listing 8.11: Type-level $>_c$ (better conversion) and $>_m$ (better member) predicates

```
type family   TupleToList t
type instance TupleToList ()        = TNil
type instance TupleToList Int32     = Int32 ::: TNil
type instance TupleToList String    = String ::: TNil
type instance TupleToList Bool      = Bool ::: TNil
type instance TupleToList Double    = Double ::: TNil
type instance TupleToList (Obj x)   = Obj x ::: TNil
type instance TupleToList (a, b)    = a ::: b ::: TNil
type instance TupleToList (a, b, c) = a ::: b ::: c ::: TNil
...

type family   ListToTuple t
type instance ListToTuple TNil                   = ()
type instance ListToTuple (a ::: TNil)           = a
type instance ListToTuple (a ::: b ::: TNil)     = (a, b)
type instance ListToTuple (a ::: b ::: c ::: TNil) = (a, b, c)
...
```

Listing 8.12: TupleToList and ListToTuple type functions

produces tuples instead of type-level lists. In this code, **TupleToList** is used to convert the tuple of argument types (*as*) to a list for the overload resolution algorithm, which ultimately returns a list of types that is converted back to a tuple with **ListToTuple**. All of this happens at the type level; the tuple, of type *as*, containing the argument values is *not* converted to a list.

```
type family Resolve as ms
type instance Resolve as ms =
  ListToTuple (FromSingleton
    (FilterBestMembers (TupleToList as)
                       (FilterApp (TupleToList as) ms)
                       (FilterApp (TupleToList as) ms)))
```

Listing 8.13: Tuple-supporting **Resolve** type function

Coercing arguments

Since overload resolution takes subtyping into account when resolving a function member, it is possible that the argument types returned by **Resolve** differ from the argument types that were provided. As with implicit conversions, we need to perform coercions on these argument values before invoking the method. We extend the **Coercible** type class defined in §8.4.2 to safely coerce *tuples* of values, instances for 0 and 2-tuples are shown in Listing 8.14.

```
instance Coercible () () where
  coerce = id
instance (Coercible f1 t1, Coercible f2 t2) ⇒ Coercible (f1, f2) (t1, t2) where
  coerce (f1, f2) = (coerce f1, coerce f2)
...
```

Listing 8.14: Extending **Coercible** to coerce tuples

8.5.4 Application

As with the implementation of implicit conversions, having an algorithm at the type-level is not enough. We need to connect the type-level overload resolution algorithm to the functions that are used to invoke methods and constructors. We show how this can be done in the context of a hypothetical method invocation. Take the following .NET class containing a single overloaded static method, Write:

```
public class Console
{
  public static void Write(String s)  {···}   // (A)
  public static void Write(Double d) {···}   // (B)
}
```

We would like to be able to invoke Write by calling a Haskell function **invokeWrite**. For example:

```
invokeWrite args = ···

main = do
  invokeWrite ("Salsa")
  invokeWrite (10 :: Int32)
```

where **invokeWrite** invokes the appropriate method implementation.

Assuming we can arrange for the overload resolution algorithm to return the signature of the chosen method as a tuple, we need to be able to dispatch the invocation to the appropriate method implementation based on this type. Haskell's type class system provides exactly what we need to implement such a type-indexed invocation function. We define a type class **WriteInvoker** which has an instance for each method implementation of the overloaded method.

```
class WriteInvoker args where
  rawInvokeWrite :: args → IO ()

instance WriteInvoker (String)  where rawInvokeWrite s  = ···
instance WriteInvoker (Double) where rawInvokeWrite d = ···
```

Connecting **invokeWrite** to **rawInvokeWrite** must involve the overload resolution algorithm and the execution of any implicit conversions.

To keep the type signatures in this example clean, we define a type synonym **WriteCandidates** that is equal to the type-level list of the signatures for the Write method:

```
type WriteCandidates = (String ::: TNil) ::: (Double ::: TNil) ::: TNil
```

We define **invokeWrite** in terms of **rawInvokeWrite** (defined above) and the **coerce** function extended to handle tuples (defined in the previous section):

```
invokeWrite :: ∀args args'.
  (Resolve      args WriteCandidates ~ args',
   Coercible    args args',
   WriteInvoker args') ⇒
   args → IO ()
invokeWrite args = rawInvokeWrite (coerce args :: args')
```

The function uses **coerce** to apply any implicit conversions to the argument values before invoking the appropriate method implementation with **rawInvokeWrite**. As with the implicit conversion support however, the magic is in the type signature. The context of **invokeWrite**'s type signature lists three constraints:

Resolve The effect of this constraint is to apply the overload resolution algorithm to determine which signature from **WriteCandidates** should be invoked given the argument types (*args*). This is expressed as an equality constraint which allows the result to be given a name, *args'*.

Coercible This constraint ensures that there is a **Coercible** instance for converting the argument tuple to the required type. Such an instance will exist if the **Resolve** constraint is satisfied.

Invoker This constraint ensures that there is a **WriteInvoker** instance for invoking the appropriate Write implementation, as identified by the method signature in *args'*.

Also of note is the use of the lexically scoped type variable *args'* in the type annotation attached to the application of **coerce**. Without this type annotation, the compiler is not aware that the result type of **coerce** should be *args'*.

Using the above implementation for **invokeWrite**, invocations of the Write method behave just as they would if invoked from a C# program. The appropriate method implementations are called for the following code, even for the second call where the Int32 argument has to be implicitly converted to a Double in order to call the (B) overload.

```
invokeWrite ("Salsa")         -- invokes (A)
invokeWrite (42 :: Int32)     -- invokes (B)
invokeWrite (pi :: Double)    -- invokes (B)
```

An extension of the technique described herein is used in Salsa for invoking methods and constructors. Since the Salsa implementation uses labels, a single function **invoke** and a single type class **Invoker** is used to handle all .NET methods and constructor invocations, rather than the method-specific implementations of this example. This label-based system is described in Chapter 7.

8.6 Error reporting

An undesirable consequence of implementing complicated algorithms at the type-level is the verbosity of the error messages produced by the compiler when something goes wrong.

If a deep type function application fails to be substituted with an instance body, then several screens of the partially evaluated type function implementation can appear on screen. For example, if a non-bridge type is used in a method application, and consequently there is no **TyCode** instance for the type, a very long error message will ensue.

On the other hand, if the overload resolution fails legitimately (due to an ambiguous invocation, for example) the type checking fails at the invocation function and results in a relatively meaningful error message like the one below:

```
No instances for (Coercible (Obj Object_)
                            (Salsa.Error Salsa.NoMatch),
                  Invoker (Salsa.Error Salsa.NoMatch))
  arising from a use of ‘invoke’ at ...
```

As more experience is gained with these sorts of applications of type functions, it should be possible to add compiler support for producing more concise error messages for these usage patterns.

8.7 Summary

This chapter demonstrated that not only is it possible to deal with both object-oriented overloading and subtyping in Haskell, but that it can be done in an elegant way that:

- is statically type-checked,
- avoids superfluous type annotations, and

- requires only general-purpose Haskell extensions (in particular, type families).

This was achieved by implementing algorithms to perform overload resolution and implicit conversion calculations — tasks that are typically performed by a C# compiler — in the Haskell type system as type functions. These algorithms were then connected, using type constraints and type classes, to the value-level infrastructure for interacting with objects.

9

Discussion

A suitable way of presenting the results of this thesis is to show Salsa in action. Consequently, this chapter presents a Haskell program that uses the Salsa library in order to interoperate with .NET.

Listing 9.1 contains the Haskell program, while Listing 9.2 shows the equivalent C# program for reference. The code speaks for itself: Salsa's goal of providing a usable, safe, convenient, lightweight and practical bridge to .NET, has been achieved.

```
module Main where

import Salsa
import Salsa.System.Windows.Forms

main :: IO ()
main = withCLR $ do
  f ← new Form_ ()
  set f [Text_ := "Saucy Salsa Sample"]
  b ← new Button_ ()
  set b [Left_  :== 10,
         Top_   :== 10,
         Text_  := "E&xit",
         Click_ :+> delegate EventHandler_
                              (λ_ _ → f # _Close ())]
  get f Controls_ ≫= # _Add (b)
  Application_ # _Run (f)
```

Listing 9.1: Salsa sample

```
using System;
using System.Windows.Forms;
class Program
{
  static void Main(string[] args)
  {
    Form f = new Form();
    f.Text = "Saucy Salsa Sample";
    Button b = new Button ();
    b.Left   =  10;
    b.Top    =  10;
    b.Text   =  "E&xit";
    b.Click += delegate(object s, EventArgs e)
    {
      f.Close();
    };
    f.Controls.Add(b);
    Application.Run(f);
  }
}
```

Listing 9.2: Salsa sample (C# equivalent)

10

Conclusion

The original goal of this thesis was to allow Haskell and .NET programs to interoperate by building a software bridge between their respective runtime systems. The development of Salsa as part of this thesis certainly satisfies the original goal.

In satisfying this goal, Salsa provides Haskell with access to the extensive collection of libraries in the .NET framework. It provides this access in a type-safe way, with a natural syntax, and without requiring arbitrary language extensions.

Quite unexpectedly however, Salsa was not the only outcome of this thesis. In overcoming the challenges of building the bridge, some useful techniques were discovered for supporting object-oriented features such as subtyping and overloading in Haskell, in a type-safe way. Properly generalised, these techniques could well be useful outside the domain of bridging Haskell with .NET, to the point of encoding arbitrary static type checking algorithms in the source code of normal Haskell programs.

10.1 Future work

Despite the development of Salsa, and the additional unexpected outcomes from this work, there is still a great deal of scope for future work. Some of this future work takes the form of continuing the development of Salsa, but there is also scope for future research work.

10.1.1 Development directions

In its current state, Salsa is a useful tool for interacting with .NET libraries, however it is not complete. There are a number of areas where the feature set of Salsa can be extended:

- Extend Salsa to support all C# features, including: arrays, explicit coercions, operators, indexers, exception handling, and additional primitive data types.
- Investigate further optimisations to the marshaling system and dynamic code generation.
- Automatically include the driver assembly into Haskell executables that use the Salsa library.
- Extend the bridge to support bidirectional interoperability, including the ability to host the Haskell runtime system in a .NET application.
- Support Mono, an open-source implementation of the .NET platform.

10.1.2 Research directions

There are a number of directions in which research work relating to this thesis could proceed, these include:

- Use type functions to statically check constraints on generic type arguments, and implement support for generics in Salsa.
- Exploring the use of type functions for encoding arbitrary static type checking algorithms at the type level, in Haskell programs.
- Providing compiler support for concise error messages in the context of type-level algorithms implemented with type functions.
- Employing rewrite rules in the Haskell compiler in order to minimise inter-runtime control transfers by grouping consecutive foreign-runtime operations together.

List of Figures

List of Listings

References

[1] Manuel M. T. Chakravarty, Tom Schrijvers, et al. GHC/Type families - Haskell Wiki. `http://haskell.org/haskellwiki/GHC/Type_families`. Accessed: 28 October 2007.

[2] Manuel M. T. Chakravarty, Gabrielle Keller, and Simon Peyton Jones. Associated Type Synonyms. In *ICFP '05: Proceedings of the International Conference on Functional Programming*. ACM Press, September 2005. URL `http://www.cse.unsw.edu.au/~keller/CP05.html`.

[3] Microsoft Corporation. .NET Framework Developer Center. `http://msdn2.microsoft.com/en-us/netframework`. Accessed: 14 May 2007.

[4] ECMA. *ECMA-334: C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2002.

[5] Sigbjorn Finne. Hugs98 for .NET. `http://galois.com/~sof/hugs98.net`. Accessed: 3 May 2007.

[6] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *International Conference on Functional Programming*, pages 153–162, 1998.

[7] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. Calling hell from heaven and heaven from hell. In *International Conference on Functional Programming*, pages 114–125, 1999.

[8] Apple Inc. The Objective-C programming language. `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf`. Accessed: 13 May 2007.

[9] Mark P. Jones. Type Classes with Functional Dependencies. *Lecture Notes in Computer Science*, 1782, 2000. URL `http://web.cecs.pdx.edu/~mpj/pubs/fundeps.html`.

[10] Mark P. Jones et al. Hugs 98. `http://www.haskell.org/hugs`. Accessed: 3 May 2007.

[11] Einar Karttunen. hs-fltk. `http://www.cs.helsinki.fi/u/ekarttun/hs-fltk`. Accessed: 28 October 2007.

[12] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2001. ACM Press.

[13] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. Draft; Submitted for journal publication; online since 30 Sep. 2004; Full version released 10 September 2005, 2005.

[14] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017488.

[15] Daan Leijen. wxhaskell. `http://wxhaskell.sourceforge.net`. Accessed: 28 October 2007.

[16] Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. Extending the Haskell foreign function interface with concurrency. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 22–32, New York, NY, USA, 2004. ACM Press.

[17] John McCarthy. *LISP 1.5 Programmer's Manual.* The MIT Press, 1962. ISBN 0262130114.

[18] Erik Meijer and Sigbjorn Finne. Lambada, Haskell as a better Java. In *Proceedings of the 2000 ACM SIGPLAN workshop on Haskell*, 2000. URL `http://research.microsoft.com/~emeijer/Papers/Lambada.pdf`.

[19] Liam O'Boyle. Making Haskell .NET compatible, 2002.

[20] André T. H. Pang. Binding Haskell to object-oriented component systems via reflection, June 2003.

[21] André T. H. Pang and Manuel M. T. Chakravarty. Interfacing Haskell with object-oriented languages. In *Implementation of Functional Languages: 15th International Workshop*, pages 20–36. Springer-Verlag, September 2004.

[22] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components in Haskell. In *Fifth International Conference on Software Reuse*, Victoria, British Columbia, 1998. URL `citeseer.ist.psu.edu/jones97scripting.html`.

[23] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[24] Joel Pobar. Reflection: Dodge common performance pitfalls to craft speedy applications. `http://msdn.microsoft.com/msdnmag/issues/05/07/Reflection/default.aspx`. Accessed: 3 May 2007.

[25] Microsoft Research. F#. `http://research.microsoft.com/fsharp`. Accessed: 8 May 2007.

[26] Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, and Manuel M. T. Chakravarty. Towards Open Type Functions for Haskell (draft). `http://www.cse.unsw.edu.au/~chak/papers/SSPC07.html`. Accessed: 24 October 2007.

[27] Mark Shields. A compiler writer's guide to C#. `http://cartesianclosed.com/pub/csharp`, May 2002. Lecture notes.

[28] Mark Shields and Simon Peyton Jones. Object-oriented style overloading for Haskell. Technical report, Microsoft Research, Cambridge, August 2001. Available at `http://www.cse.ogi.edu/~mbs/pub/overloading/overloading.ps`.

[29] Axel Simon, Duncan Coutts, et al. Gtk2hs. `http://www.haskell.org/gtk2hs`. Accessed: 28 October 2007.

[30] Wolfgang Thaller and André Pang. Private communication, October 2003.