Space Cost Analysis Using Sized Types

A thesis to be submitted to the
UNIVERSITY OF ST ANDREWS
for the degree of
DOCTOR OF PHILOSOPHY

by
Pedro Baltazar Vasconcelos

School of Computer Science
University of St Andrews

25th August 2008

I, Pedro Baltazar Vasconcelos, hereby certify that this thesis, which is approximately 65,000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date ____________________ *signature of candidate* ____________________

I was admitted as a research student in September 2002 and as a candidate for the degree of Doctor of Philosophy in September 2002; the higher study for which this is a record was carried out in the University of St Andrews between 2002 and 2008.

date ____________________ *signature of candidate* ____________________

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date ____________________ *signature of supervisor* ____________________

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker, that my thesis will be electronically accessible for personal or research use, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration.

date ____________________ *signature of candidate* ____________________

Abstract

Programming resource-sensitive systems, such as real-time embedded systems, requires guaranteeing both the functional correctness of computations and also that time and space usage fit within constraints imposed by hardware limits or the environment. Functional programming languages have proved very good at meeting the former *logical* kind of guarantees but not the latter *resource* guarantees.

This thesis contributes to demonstrate the applicability of functional programming in resource-sensitive systems with an automatic program analysis for obtaining guaranteed upper bounds on dynamic space usage of functional programs.

Our analysis is developed for a core subset of *Hume*, a domain-specific functional language targeting resource-sensitive systems (Hammond et al. 2007), and presented as a type and effect system that builds on previous sized type systems (Hughes et al. 1996, Chin and Khoo 2001) and effect systems for costs (Dornic et al. 1992, Reistad and Gifford 1994, Hughes and Pareto 1999). It extends previous approaches by using abstract interpretation techniques to *automatically* infer linear approximations of the sizes of recursive data types and the stack and heap costs of recursive functions.

The correctness of the analysis is formally proved with respect to an operational semantics for the language and an inferrence algorithm that automatically reconstructs size and cost bounds is presented.

A prototype implementation of the analysis and operational semantics has been constructed and used to experimentally assess the quality of the cost bounds with some examples, including implementations of textbook functional programming algorithms and simplified embedded systems.

Acknowledgement

First and foremost I would like to thank my supervisor Prof. Kevin Hammond for proposing a challenging problem and providing me with the opportunity to conduct research in a stimulating environment at the University of St. Andrews. My thanks extends to Prof. Roy Dyckhoff and Dr. James McKinna for engaging in many interesting discussions on logic and type theory.

I thank Prof. Roberto Bagnara for his help in using the Parma Polyhedra Library and for inviting me to spend a week in January 2005 visiting the Department of Mathematics at the University of Parma. I also thank Axel Simon for providing me with his experimental bindings for using the Parma Polyhedra Library from Haskell, and the members of my viva panel, Dr. Graham Hutton and Dr. Mike Livesey, for numerous helpful suggestions for improving this thesis.

Furthermore, I thank the UK government's Engineering and Physical Sciences Research Council for the scholarship that supported my study in St. Andrews.

Finally, I thank my wife Sofia without whose enduring support this thesis would not have been possible.

Declaration

Earlier versions of the size and cost analyses presented in this thesis were developed in joint collaboration with other researchers; this work appears in (Portillo, Hammond, Loidl and Vasconcelos 2003, Vasconcelos and Hammond 2004) and is reviewed in the literature survey of Chapter 3. However, all technical material in Chapters 5 and 6 of this thesis (the abstract machine, type inference rules, soundness proofs, reconstruction algorithm and experimental results) is new.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we present an overview of this thesis: we outline our research problem, namely, obtaining guaranteed bounds for dynamic space consumption of embedded-systems programmed in a functional language; we define our methodology, namely, the development of a type-based program analysis; and we illustrate the approach on a short example (a digital filter). Finally, we highlight the contributions and the structure of this thesis.

1.1 Motivation

Programming resource-sensitive systems, such as real-time embedded systems, requires guaranteeing both the functional correctness of computations and also that time and space usage fit within constraints imposed by hardware limits or the environment. Functional programming languages have proved very good at meeting the former *logical* kind of guarantees but not the latter *resource* guarantees.

This thesis aims at extending the applicability of functional programming languages to resource-sensitive systems by developing an automatic program analysis to provide upper bounds on the time and space upper usage required for functional programs. While such guarantees are particularly important in embedded or safety-critical systems, they are are also useful in other resources-sensitive settings, e.g. in environments supporting the execution of untrusted code received from a network.

The standard computability arguments imply that no algorithm can provide resource bounds for *all* programs of sufficient expressiveness (functional or otherwise). This thesis contributes with a partial solution, namely, a programming language subset and program analysis capable of obtaining guaranteed upper bounds for stack and heap costs of non-trivial programs. The correctness of the analysis is formally proved with respect to an operational semantics for the language and the quality of the bounds obtained are experimentally verified against some examples, including implementations of textbook functional programming algorithms and prototype embedded systems.

1.2 Resource-sensitive systems

Software is increasingly used in systems that are not general-purpose computers and whose primary role is to interact with the environment, e.g. in a microprocessor-controlled washing machine, in the control of an industrial robot or in an aircraft flight controller. Such software is generically designated as *embedded*, to signify that it is part of a larger engineering system which it must support (Burns and Wellings 1996).

Embedded software must satisfy both functionality requirements (that responses are logically correct) and resource guarantees (that responses are computed within fixed bounds on time, dynamic memory or energy consumption). In order to meet the latter non-functional requirements while making efficient use of available hardware, embedded systems were traditionally programmed in low-level assembly languages.

The increase in complexity of applications means that it is no longer cost-effective to develop embedded systems solely in assembly language. Today's embedded systems are typically developed in higher-level languages such as C, C++, Ada or even Java with only a very limited part in assembly code. The use of higher-level languages eases the detection and correction of errors, facilitates portability and re-use of code and generally increases the speed of development. The price of such facilities is a loss of predictability of the time and space usage of programs.

For applications that are not critical, the loss of predictability can be mitigated during testing, e.g. by profiling time and space usage. Testing, however, is very time-consuming and the results can be invalidated by even minor code changes during development. Moreover, as pointed out by Dijsktra (1970) more than 30 years ago, testing can show the *presence* but never the *absence* of errors.

The current software development practice for high-integrity systems is to prohibit language features that may lead to unpredictable resource usage: the SPARK subset of the Ada language excludes recursion and dynamic memory allocation (Finnie and Amey 2006); the real-time kernel of the ARIANE 5 software was also verified to exclude dynamic memory allocation (Lacan et al. 1998). However, this is done at the loss of useful programming techniques and abstractions. For example, Stankovic (1988) points out that dynamic memory allocation is essential for the next-generation of embedded systems. As embedded applications become more complex, there is an increasing interest in techniques that combine programming languages, formal methods and implementations to increase the level of programming while still guaranteeing safety properties of the executed code.

A note on terminology Embedded systems are designated *real-time* if their correctness depends on guarantees that outputs are produced within strict time deadlines. These are further divided into *hard real-time* and *soft real-time* according to the level of criticality: a late response is undesirable but still tolerable in soft real-time systems; in hard real-time systems a late response is not acceptable. For example, decoding an audio stream in a digital audio player is a soft real-time task, while the flight control of a commercial aircraft is a hard real-time task.

Some researchers identify the terms "embedded" and "real-time" and use them as synonyms (Burns and Wellings 1996); this is perhaps because resources such as space could be trivially bounded in traditional embedded systems programming, e.g. using static allocation. However, the desire to use dynamic memory management techniques in embedded programming means that space bounds are also relevant. We therefore prefer to use "embedded" for generic resource-sensitive systems and "real-time" for the time-sensitive ones.

1.3 Functional programming

Functional programming is a style for writing programs where the principal mechanism of computation is the evaluation of expressions. A functional program consists of equations defining mathematical transformations of inputs to outputs rather than a sequence of commands that modify the internal state of the machine as in a typical imperative language such as C, C++ or Java.

Modern functional programming languages, e.g. Standard ML (Milner et al. 1997), Objective Caml (Leroy et al. 2007) and Haskell (Jones 2003) have evolved to include several features that support the functional programming style:

Functions are first-class values: Functions can be passed as arguments to other functions, returned as results, or stored in data structures.

Strong type system: Values are classified into *types*; moreover, there is no "void" type that can be used in absence of a meaningful type for an expression; this allows an interpreter or compiler to reject some erroneous programs automatically by checking that types are used consistently; furthermore, for the most part, type annotations can be automatically inserted by the compiler using *type inference.*

Parametric polymorphism: Types may be parameterised over other types; this allows a function that can be used with values of many types to be assigned a *universally quantified type.* The distinct notion of *ad-hod polymorphism*, or *overloading*, concerns using the same names for different implementations according to context, e.g. numerical operators; the connection between the two kinds of polymorphism is made via *type classes* (Wadler and Blott 1989) and is used in Haskell.

Definitions by equations: Functions can be defined by equations; simple equational reasoning can be used to derive new programs from old ones or to prove properties.

Algebraic data types: New data types can be defined as a disjoint sums of products labelled by *constructor tags*; such definitions can also be *recursive*; case analysis with *patterns* can be used to discriminate constructed values and bind variables.

Non-strict semantics: Under non-strict semantics an expression is only evaluated when its result is needed to progress the computation; the evaluation strategy is therefore driven by data dependencies rather than by the syntactical structure of the program. While not a universal characteristic of functional languages[1], non-strict semantics allows for more compositional programs by separating the producers and consumers of data structures (Hughes 1989); in particular, it allows the manipulation of potentially infinite data structures.

Automatic memory management: Allocation and deallocation of memory is performed automatically by the runtime system; this not only avoids the need for the programmer to introduce explicit commands for memory management but also eliminates some errors arising from incorrect uses, e.g. accessing "dangling" references.

The higher conceptual level of functional languages allows writing programs that are more modular, easier to modify and prove correct. Indeed, programming in functional languages often resembles writing "executable specifications" of the underlying problem or domain (Hudak and Jones 1994, Jones et al. 2000). The downside is that

[1] Haskell has a non-strict semantics, but Standard ML and O'Caml do not.

the implementation has to bridge a larger gap between the functional program and the underlying hardware than that of a typical imperative language.

The functional programming community has made large progresses in optimising compilers and profiling tools that enable functional programs to match imperative ones.[2] In the opposite direction, many language features that originated in functional languages have migrated to mainstream imperative languages such as Java, e.g. automatic memory management, static type checking and polymorphism. This means that the performance gap has narrowed and is arguably no longer the most relevant argument for preferring one language paradigm over the other (Wadler 1998*b*).

For the specific domain of embedded programming, however, the situation is quite distinct: efficiency is important, but the foremost concern is not absolute performance but rather *predictability* of performance (Stankovic 1988). Functional languages present particular difficulties because their design abstracts away from the non-functional characteristics that need to be assured, e.g. time and space costs. We identify some issues that make it more difficult to reason about the performance of functional programs:

Higher-order functions: The control-flow of higher-order programs is dynamic hence more difficult to determine; functional values are implemented as *closures* in memory with associated space and time costs.

Data persistence: Purely-functional data structures are *persistent* (Okasaki 1998), that is, an update does not modify existing data but instead creates a new version that coexists with the old one. Even when the data is used in a single-threaded way, it is difficult to ensure that the implementation will immediately re-use the old space. By contrast, the update-in-place typical of imperative data structures is sufficient, in many cases, to ensure bounded space computation.

Garbage collection: This common technique for automatic memory management can cause time delays at arbitrary points during execution while the collector reclaims unreachable data; furthermore, collectors are usually conservative, that is, they may fail to reclaim all unreachable data, leading to the possibility of memory leaks and making it more difficult to predict actual memory residency.

[2] For example: the Haskell GHC and O'Caml benchmarks in the *Computer Language Benchmarks Game* (`http://shootout.alioth.debian.org`) as of February 2008 are, on average, only 50%–70% slower than the best (imperative) solution and rank better than many other imperative ones e.g. Java.

Non-strict semantics: *Non-strict* (or *lazy*) evaluation means that the time and space costs for a function depend not just on its inputs also on the *context* in which the result is used; lazy evaluation can also contribute to space leaks because suspended computations ("thunks") can contain references to data objects making them live longer than necessary.

Optimising compilers: The highly declarative nature of functional programs means that compilers are free to perform various transformations to improve performance; however, it is difficult to be sure at the source level which optimisations are performed and what their impact will be on the time and space behaviour.

Such difficulties have prevented the use of functional languages in applications requiring hard bounds on space and time response. We will address some of these difficulties in this thesis.

1.4 Aim and methodology

The overall aim of this thesis is to demonstrate the feasibility of combining the high-level features of functional languages with the strict resource guarantees required by embedded systems. Given the difficulties in reasoning about time and space costs outlined in the previous section, we propose the following methodology:

- *We focus on Hume, a functionally-based language for programming resource-sensitive systems* (Hammond et al. 2007). Unlike a general-purpose functional language, Hume is designed to allow predictable time and space behaviour, both at the language level, e.g. by syntactically separating the *coordination* and *expression* layers of computation, and in the implementation, e.g. by not requiring a garbage collector.

- *We further restrict to a "core" subset of Hume.* This simplifies the presentation of the language by omitting redundant features that can be obtained by translation (e.g. nested pattern matching). It also facilitates the formal definition of a model of execution costs (an *abstract machine*) and makes the relation between language and execution amendable to formal reasoning.

- *We develop a type-based static analysis for predicting bounds on dynamic space costs.* Types are pervasive in modern functional programming; adding analysis information onto types is, therefore, a natural extension. Morever, unlike other approaches that require the whole program (e.g. data-flow or abstract interpretation), type-based analyses suport *modularity*, i.e. gathering information about libraries and modules separate from its uses.

Fulfilling these points will provide us with the combination of a functional core language, execution model and static analysis that guarantees safe bounds on resources. This paves the way for using high-level abstractions of functional programming languages in embedded systems whenever they can be statically proved safe rather than prohibiting them in all situations.

At this point we would like to clarify what we mean by an "automatic analysis" for resources. Every computer science student learns that the Halting Problem is algorithmically unsolvable, i.e. that there is no algorithm that decides if an arbitrary program in a Turing-complete language terminates. Since providing a time bound entails proving termination, it follows that there can be no algorithm that provides an upper bound on computation time for an arbitrary program in a Turing-complete language.[3]

There are two ways out of this impossibility: either 1) consider a restricted language that is *not* Turing-complete; or 2) allow a *partial* bounding algorithm, i.e. one that may yield a "don't know" answer for some programs. In this thesis we will follow the second option, that is, we will consider a Turing-complete language but allow the analysis to give uninformative answers.

A second question is what we mean by "guaranteeing bounds" on resources. It is immediate that every finite (that is, terminating) computation can only consume a finite amount of time and space; thus a termination proof of a program or function in a Turing-complete language also proves that it is bounded. In this "extensional" sense, any total program is bounded because we can simply run it under an instrumented interpreter to find out the amount of resources it consumes. However, this does not give a satisfactory static analysis—it amounts to profiling the program for each possible input.

What we desire are resource bounds expressed as the composition of simple mathematical functions with known growth, e.g. polynomials, rational functions, exponentials, etc. Moreover, we would like to obtain these expressions not as functions of the input itself but rather of some abstraction such as data size. Such bounds are useful for a programmer to understand the behaviour of his or her program, but also for a verification system to automatically ensure *before* execution that an embedded system will not run out of memory or miss a deadline.

[3] More precisely: a function from Turing-machine descriptions to the number of steps until termination (arbitrarily defined in the case of non-termination) grows faster than *any* computable function (Minsky 1967, pages 146–148).

1.5 An example: a digital filter

We will illustrate the Hume language approach to embedded systems programming and our methodology for obtaining cost bounds using a simple example of an idealised digital filter.

Consider an infinite stream x_0, x_1, $x_2, \ldots$ of real-valued samples; a *finite impulse response filter* (Pohlmann 1989), or simply *FIR filter*, has an output stream y_n satisfying equation (1.1)

$$y_n = w_0 \times x_n + w_1 \times x_{n-1} + \cdots + w_{k-1} \times x_{n-k+1} \tag{1.1}$$

where $w_0, \ldots, w_{k-1}$ are the filter coefficients and $k > 0$ is the filter order, i.e. the number of previous input samples required to compute one output sample.

1.5.1 Solutions in Haskell and Hume

We can program the FIR filter in a general purpose functional language using lazy lists to represent potentially infinite streams of inputs and outputs. Figure 1.1 presents an implementation in Haskell. The input and output stream are modelled as infinite Haskell lists; the filter itself is a corecursive function that maps an input stream to an output stream of floating-point samples.[4] We use the generic list functions *tail*, *map*, *zipWith*, *iterate* and *sum* from the standard Haskell prelude (Jones 2003).

Although it is apparent from equation (1.1) that only k input samples are required to compute each output, there is no guarantee that the Haskell execution will deallocate the previous samples as each output is produced. Moreover, even if the garbage collector does eventually recycle space, no guarantee can be given *from the program text* about how much of the input stream is be retained in memory.

This unpredictable behaviour is avoided in Hume by making the corecursive level of computation explicit as a finite network of processes, called *boxes*. Each box has a fixed number of ports and runs indefinitely mapping streams of inputs to outputs. Computation in each box is specified using a strict, purely-functional notation. Furthermore, all state information required by a box must be explicitly represented as an input-output feedback loop; this ensures that all memory can be re-used at the end of a box execution cycle.

[4] *Corecursion* is a construction principle for infinite data structures, e.g. streams. Whereas recursion deconstructs a well-founded (i.e. finite) data structure, corecursion builds up an infinite *codata* structure (Turner 2004). Denotationally, the two notions are dual: recursive definitions correspond to *least-fixed points* of a monotone operator, while corecursive ones correspond to *greatest-fixed points* (Barwise and Moss 1996).

```
import Prelude
-- tail :: [a] -> [a]
-- map :: (a->b) -> [a]->[b]
-- iterate :: (a->a) -> a -> [a]
-- zipWith :: (a->b->c) -> [a] -> [b] -> [c]
-- sum :: [Float] -> Float

-- FIR filter coeficients: (0.5, 2, 0.5)
fir :: [Float] -> [Float]
fir stream = map (dotp [0.5,2,0.5]) (iterate tail stream)

dotp :: [Float] -> [Float] -> Float   -- dot product of two lists
dotp xs ys = sum (zipWith (*) xs ys)
```

Figure 1.1: FIR filter example in Haskell.

The Hume solution for the FIR filter is presented in Figure 1.2 as a single box with two inputs and two outputs: x is an input sample, xs is a list with the last k samples y is the filtered output and xs' is the updated list of samples.

The wire declaration establishes a feedback loop between an input and output of the box; this is the sole mechanism in Hume for passing state information across box iterations. The feedback wire between xs and xs' in the filter example communicates the list of previous k samples. Note that lists can have arbitrary size; the fact that the list in the wire xs, xs' has bounded length is a dynamic property of this particular program that we can determine by static analysis.

We use two auxiliary functions: *init* gives the initial segment of a list and *dotp* computes the dot product of two lists of samples (as in the Haskell solution). The Hume version is defined using first-order recursion to make it amendable to our cost analysis.

1.5.2 Size analysis

In order to implement the Hume program in bounded space we need to guarantee that the input xs and output xs' are both bounded. This entails establishing size bounds on a list (more generally, on any input or output of a box associated with a recursive data type); we will employ a type-based size analysis for this purpose.

From the recursive definition of *init* in Figure 1.2, our size analysis will automat-

```
type Float = float 32  -- 32-bit floating point numbers

box fir
in (x::Float, xs::[Float])
out (y::Float, xs'::[Float])
match
  (x, xs) -> (dotp [0.5,2,0.5] xs, x:init xs)
wire fir.xs fir.xs' initially [0,0,0]

dotp :: [Float] -> [Float] -> Float   -- dot product
dotp []     []     = 0
dotp (x:xs) (y:ys) = x*y + dotp xs ys

init :: [a] -> [a]  -- initial segment of a list
init [x] = []
init (x:xs) = x:init xs
```

Figure 1.2: FIR filter example in Hume.

ically infer the following annotated type:

$$\mathit{init} : \langle \mathsf{List}^n\, a \rightarrow \mathsf{List}^m\, a,\ n = 1 + m \wedge 0 \leq m \rangle \tag{1.2}$$

The data type $\mathsf{List}^n\, a$ for lists of a is annotated with a *size variable* n representing the length of the list; the type signature as a whole is annotated with a constraint that expresses *size relations*. The constraint in (1.2) expresses the input-output size relation, namely that the output list has one less element than the input list.[5]

We can use (1.2) to derive a sized type for the expression computing the output xs':

$$\overbrace{\underbrace{x}_{\mathsf{Float}} : \overbrace{\mathit{init}\ \underbrace{xs}_{\mathsf{List}^n\ \mathsf{Float}}}^{\mathsf{List}^m\ \mathsf{Float}}}^{\mathsf{List}^n\ \mathsf{Float}} \qquad \text{where } n = 1 + m$$

The above sized type proves that the length of the buffer of samples is invariant (as would be expected, since one element is removed for each new one inserted).

[5] Although we could present this particular type more succintly, e.g. as $\mathsf{List}^{1+m}\, a \rightarrow \mathsf{List}^m\, a$ for $m \geq 0$, such simplifications do not generalise, e.g. when the size constraint expresses inequalities rather than equalities or when it envolves more than two variables.

1.5.3 Cost analysis

We want not just to prove that the FIR filter is bounded but also to statically obtain bounds on the actual time and space costs. Unlike the size information obtained in the previous section, such properties are not denotational but rather dependent on a implementation; this means that we must choose some execution model to reason about costs. In this thesis we consider dynamic space costs, i.e. *stack* and *heap* required by an abstract machine; the machine and cost model will be formally defined in Chapter 6; for the moment, it suffices to say that it is based on the well-known SECD machine (Landin 1964).

Our cost analysis is expressed by types annotated with both sizes and *effects* representing bounds on the stack and heap costs. For the *init* function in Figure 1.2 the analysis automatically infers the following:

$$\mathit{init} : \langle \mathsf{List}^n\, a \xrightarrow{s;h} \mathsf{List}^m\, a,\ n = 1 + m \wedge 0 \leq m \wedge s \leq 6n - 3 \wedge h = 3n - 2\rangle \qquad (1.3)$$

The function type is annotated with two variables, s and h, that represent the stack and heap costs of the function body, and a constraint expressing the costs relative to the sizes of inputs or outputs, i.e. the list lengths; this mechanism allows expressing costs that depend on the inputs such as those of recursive functions.

Informally, the annotated type (1.3) means that evaluating *init* of a list of n values requires $6n - 3$ words available on the stack and will consume exactly $3n - 2$ heap cells; these results are as would be expected since *init* is defined by recursion on the input list and must construct a new list with one fewer value; the constants in the cost equations are a consequence of the particular data representations of our abstract machine.

Note that the *init* function is undefined for the empty list and this is expressed in the annotated types (1.2) and (1.3): the constraint $n = 1 + m \wedge m \leq 0$ restricts valid input list lengths to be strictly positive. We therefore do not need to consider the case $n = 0$ in the stack and heap equations in (1.3).

Note also that (1.3) expresses both size and cost information. In fact, we will conduct a combined size and cost analysis rather than two separate ones. For the purpose of presentation, however, we will first focus on the size analysis and later extend it with cost information.

Applying the size and costs analysis to the *dotp* function of Figure 1.2 we obtain:

$$\mathit{dotp} : \langle (\mathsf{List}^n\, \mathsf{Float},\, \mathsf{List}^m\, \mathsf{Float}) \xrightarrow{s;h} \mathsf{Float},\ n = m \wedge s \leq 1 + 7n \wedge h = 2 + 4n\rangle \quad (1.4)$$

The annotated type (1.4) expresses not just bounds on the stack and heap costs but also that *dotp* is only defined for input lists of equal length. Note also that *dotp* consumes a linear amount of heap on the length of its inputs; this because we are assuming an implementation using heap allocated floating point numbers, i.e. a "boxed" data representation.

1.5.4 Type and cost polymorphism

Annotated types can be universally quantified on type, size and cost variables; for simplicity we have so far omitted the quantifiers. For example, the type for *init* with explicit quantifiers is:

$$\begin{aligned} \mathit{init} : \forall n, m, s, h.\, \langle \forall a.\, \mathsf{List}^n\, a \xrightarrow{s;h} \mathsf{List}^m\, a,\, n = 1 + m \wedge 0 \leq m \\ \wedge\, s \leq 6n - 3 \wedge h = 3n - 2 \rangle \end{aligned} \tag{1.5}$$

Quantification over the size and cost variables n, m, s, h allows using the function in different contexts; the input and output sizes as well as costs can be different for each use. Quantification over the type variable a means that *init* can be used with lists of any type, i.e. it is polymorphic on the elements of the list; this is just the usual parametricity result, i.e. that *init* does not scrutinise the list elements and therefore must behave uniformly with lists of arbitrary values.

However, the combination with the quantification on costs s, h implies a stronger property, namely, that the stack and heap costs of *init* are also uniform; this is the case for a naive implementation of lists as linked cells of pointers (or pointer-sized objects), but not for an optimised implementation that specialises list representations, or that fuses producer and consumer functions such as *init* and *dotp* and therefore avoids building the list.

This does not mean that our cost analysis is only applicable to naive implementations; rather, it means that optimisations must be made explicit at the source level and the analysis extended accordingly. We will exemplify this approach in Chapter 6 for optimisations of tail-calls, data type unboxing and explicit deallocation.

Finally, we remark that type and cost polymorphism gives a natural mechanism for conducting separate analysis of modules or libraries: the augmented type (1.5) captures the size and cost analysis of *init* and can be performed without prior knowledge of its uses.

1.5.5 Analysis of the coordination layer

We now illustrate how the size and costs analysis of the previous functions can be employed to obtain bounds on the costs for producing each output and for the communication buffers (in this example, the feedback loop with a list of samples).

$$
\begin{aligned}
&z_0 \sqsupseteq (z_2 = 3 \wedge z_0 \geq 6 \wedge 27 \geq z_0),\ z_0 \sqsupseteq (z_0 = 5 \wedge z_2 = 3),\\
&z_0 \sqsupseteq (z_0 = 4 \wedge z_2 = 3),\ z_1 \sqsupseteq (z_2 = 3 \wedge z_1 = 30),\\
&z_4 \sqsupseteq (z_4 \geq 6 \wedge z_2 \geq 1 \wedge 6 + 4z_2 \geq z_4),\\
&z_4 \sqsupseteq (z_4 \geq 5 \wedge z_2 \geq 1 \wedge 1 + 6z_2 \geq z_4),\\
&z_4 \sqsupseteq (z_4 = 4 \wedge z_2 \geq 1),\ z_5 \sqsupseteq (4 + 8z_2 = z_5 \wedge z_2 \geq 1),\\
&z_3 \sqsupseteq (z_2 = z_3 \wedge z_2 \geq 1),\ z_4 \sqsupseteq z_4 = 2,\ z_4 \sqsupseteq z_4 = 1,\ z_5 \sqsupseteq z_5 = 16,\\
&z_3 \sqsupseteq z_3 = 3,\ z_6 \sqsupseteq z_6 = 1,\ z_7 \sqsupseteq z_7 = 2,\ z_2 \sqsupseteq z_2 = z_3
\end{aligned}
$$

Wire	Size	Stack	Heap
$\mathit{fir}.x$: Float	–	[1, 1]	[2, 2]
$\mathit{fir}.y$: Float	–	[4, 27]	[30, 30]
$\mathit{fir}.xs$, $\mathit{fir}.xs'$: $\mathsf{List}^{z_2}\ \mathsf{Float}$	[3, 3]	[1, 19]	[16, 28]

Table 1.1: Coordination layer analysis of the Hume filter.

The coordination layer analysis extracts a set of cyclic inequations expressing lower bounds on the sizes and costs of boxes (these will be formally defined in Chapter 6). The inequations can then be automatically solved using fixed point approximation techniques to obtain intervals of sizes and stack and heap costs for the execution of individual boxes and the communication wires.

The results of the coordination layer analysis are intervals $[l,\, u]$ where $l \in \mathbb{Z} \cup \{-\infty\}$, $u \in \mathbb{Z} \cup \{+\infty\}$ and $l \leq u$ for both sizes and costs. While we are mostly interested in upper bounds of costs, lower bounds of sizes convey useful information and can improve the precision of the analysis.

Table 1.1 presents both the constraints and the solutions obtained by our analysis for the filter example. The upper bounds of the intervals of stack and heap costs can be used by a compiler to allocate memory statically with the assurance that failure due to memory exhaustion will not occur.

1.6 Contributions

The main contributions of this thesis are in the area of static analysis of space costs for functional programs; in order of relevance:

1. *A formal type-based automatic analysis for inferring bounds on sizes, stack and heap costs of Core Hume programs* (a first-order, strict functional language with recursive data types and functions). This analysis extends previous approaches by automatically *inferring* bounds rather than just *checking*

prescribed bounds.

2. *A formal abstract machine for Core Hume* that provides the model of stack and heap costs against which we validate the analysis. This machine is novel in employing region-based allocation to implement predictable memory recycling; this is better suited for reasoning about costs at the language level than existing implementations, e.g. the prototype Hume abstract machine (Hammond 2003), because every transition performs a strictly bounded amount of computation; in particular, our abstract machine does *not* employ a copying collector for recycling memory.

3. *Extensions to the abstract machine and cost analysis for space optimisations*, namely: tail calls, unboxed data representations and explicit deallocation. Dealing with optimisations is important not just for general efficiency concerns, but also for *predictability*, because optimising a program can lower the space behaviour making it tractable by our analysis.

4. *The definition of language-based cost annotations and lifting transformations.* Cost annotations decouple the source-level analysis from the precise cost model used; this is important from both a conceptual point-of-view but also because it allows trading precision for a faster analysis.

5. *Experimental results assessing the quality of the cost bounds against actual costs* obtained using a profiling implementation of the abstract machine. Our examples include standard functional algorithms on lists and trees and some prototype embedded systems.

This thesis also contributes to the area of size type analysis (Hughes et al. 1996, Chin and Khoo 2001):

1. We extend previous approaches with *user-defined sizes*; this makes it possible to infer sizes (as well as costs) for algorithms on non-linear data structures, e.g. binary trees;

2. We identify an error in the soundness proof of Chin and Khoo (2001) in assuming completeness of the lattice of constraints and present a revised proof for our sized type system.

Finally, this thesis contributes to research in the Hume language with the *formal definition of a core subset and abstract machine.* The Core Hume language and abstract machine can form the basis for an intermediate compiler language for Hume programs where memory management costs are explicit rather than delegated to the runtime system; this would be better suited for ultimately extending the analysis to provide time bounds required for real-time embedded systems programming.

This thesis only partially achieves the aim outlined in Section 1.4; three important restrictions were made:

- we focus on a subset of the Hume programming language (*Core Hume*); the principal semantic restriction is that Core Hume is a *first-order language*;
- we develop an analysis that predicts *dynamic space*, e.g. stack and heap usage, but not time;
- our analysis is restricted to obtaining size and cost bounds expressible as *linear arithmetic constraints.*

The restriction to linear arithmetic constraint is motivated by the desire for an *automatic* analysis: the linear fragment of arithmetic is decidable and implementations of efficient solvers are available (Pugh 1992, Bagnara et al. 2006); these will enable us to employ fixed-point approximation techniques to automatically infer size and cost equations for recursive functions.

The restriction to space is methodological: our approach to cost analysis is language-based, that is, we will define the analysis and execution model in a syntax-directed way. For the latter we will employ an *abstract machine* which avoids excessive details of a real implementation and is more amendable to formal proofs. The costs derived from the abstract machine should be transposable to any concrete realisation by a suitable choice of storage units.

It would, of course, be possible to consider time costs by simply counting transitions of our abstract machine and develop the analysis accordingly; however, the assumption of uniform time for machine transitions is likely not to correlate reliably with real-time and a more precise model of time would inevitably require many more implementation details.

We shall therefore pursue an analysis for ensuring bounds on dynamic space costs only; this is arguably a more fundamental problem: all resource-sensitive systems require bounded space whereas only real-time systems require static time guarantees. We also will discuss possible extensions for extending our analysis to infer realistic timing information in Chapter 8.

1.7 Organisation of this thesis

The remaining chapters of this thesis are organised as follows:

Chapter 2 reviews some background on static program analysis, focusing on two approaches used in Chapters 5 and 6, namely, *type and effect systems* and *abstract interpretation*. This chapter reviews how are analyses specified and how is their correctness established with respect to some semantic description of the programming language.

Chapter 3 reviews previous work in determining bounds for time and space usage of functional programs. These are split into separate approaches: automatic complexity analysis (Section 3.1), type and effect systems for time (Section 3.2), sized types (Section 3.3), dependent types (Section 3.4), amortised analysis (Section 3.5) and other related work (Section 3.6).

Chapter 4 reviews *Hume*, a functionally-inspired language for programming embedded systems and presents the *Core Hume* subset that will be subject of our size and cost analysis. We formally describe the syntax of the expression and coordination layers of the language, the underlying type system and the denotational semantics for expressions; the semantics of the coordination layer is delayed until Chapter 6.

Chapter 5 presents a size analysis for Core Hume programs based on a sized type system; this purely denotational size information is a preliminary step towards obtaining cost bounds for recursive programs. The size analysis is proved sound against the denotational semantics of Chapter 4. We also present an algorithm for reconstructing sized types automatically and dicuss limitations of the analysis regarding quality of the size information obtained.

Chapter 6 presents operational semantics for expressions using an *abstract machine* and for processes (boxes) using a transition system. These operational semantics serve as a formal model for space costs of programs. We then extend the size analysis of Chapter 5 with *cost annotations* and *cost effects* modelling these space costs. The soundness of the analysis is formulated with respect to a denotational semantics instrumented with costs. Finally, we present extensions for common space optimisations and show how to extend the analysis for the coordination layer.

Chapter 7 presents an experimental assessment of the cost analysis of Chapter 6 comparing predicted costs and actual costs obtained from execution profiles. We consider some textbook functional algorithms on lists and trees and some prototype embedded systems.

Chapter 8 summarises our results, the limitations of our approach and presents some directions for further research.

Chapter 2

Program analysis

In this chapter we review some of the background on techniques for static program analysis, that is, for computing sound approximations of the dynamic behaviour of programs. We focus on the two approaches that we will employ in Chapters 5 and 6, namely *type and effect systems* and *abstract interpretation*, and review how are analyses specified and how is their correctness established with respect to the semantics of the programming language.

2.1 Overview

Program analysis concerns the study of automatic techniques for obtaining predictive information about the dynamic behaviour of programs. The usual requirement is that program analysis should obtain *sound* information with respect to the program semantics, that is, obtain *approximations* that hold for all executions. This means that any approximation must be conservative, i.e. err by over-estimation of the dynamic behaviour.

The traditional motivation for program analysis is to gather information for enabling optimisations in compilers. More recently, program analysis has been applied for verifying that software respect both *safety properties* ("something bad will not happen") and *liveness properties* ("something good will happen") (Owicki and Lamport 1982). Applications of program analysis in this context include aiding detecting errors, validating software received from sub-contractors, allowing execution of foreign code in an untrusted environment and aiding in transformations of data formats (e.g. the Y2K problem).

A complete survey of program analysis is beyond the scope of this thesis; we refer the reader to the textbook of Nielson, Nielson and Hankin (1999) for a comprehensive presentation of the area.

2.2 Type and effect systems

Type and effect systems are program analysis that extend types with annotations describing properties of values or evaluations (Nielson and Nielson 1999).

Analyses based on effects were first introduced to control the combination of imperative features with functional languages (Lucassen 1987, Lucassen and Gifford 1988, Talpin and Jouvelot 1994); in this setting, *effects* are abstract descriptions of impure side-effects occurring during evaluation, e.g. accesses to imperative references or input-output actions. Other uses of type and effects analysis include exception tracking, inferring region annotations (Talpin and Jouvelot 1992, Tofte and Birkedal 1998), analysing communication in concurrent systems (Amtoft et al. 1999) and predicting execution costs (Dornic et al. 1992, Reistad and Gifford 1994, Hughes and Pareto 1999); the latter will be reviewed in detail in Chapter 3.

2.2.1 A simply-typed language

For concreteness we will consider an analysis for tracking exceptions raised during evaluation of a simple applicative language; our presentation is based on Nielson et al. (1999, chap. 5). The syntax of terms is the simply-typed lambda-calculus extended with constants, conditionals and exception raising and handling:

$$\begin{aligned} e \quad ::= \quad & c \ \mid\ x \ \mid\ \lambda x.\, e \ \mid\ (e_1\, e_2) \\ & \mid\ \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \ \mid\ \mathsf{raise}\ \epsilon \ \mid\ \mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2 \end{aligned} \tag{2.1}$$

Exceptions are identified by tokens ϵ taken from some finite set; they can be raised by the evaluation of a raise expression and handled by the expression '$\mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2$'; the latter evaluates to e_2 *unless* the exception ϵ is raised, in which case it evaluates to e_1. For simplicity, we have omitted primitive operations and recursive function definitions; the extension of the analysis for the latter is presented in Nielson et al. (1999, chap. 5).

The semantics of expressions is given in Table 2.1 by a call-by-value big-step evaluation relation $e \longrightarrow v$ meaning that expression e evaluates to the value v; values are a proper subset of expressions, namely: constants, (closed) lambda-abstractions or raised exceptions.

$$v \quad ::= \quad c \ \mid\ \lambda x.\, e \ \mid\ \mathsf{raise}\ \epsilon \tag{2.2}$$

$$c \longrightarrow c$$

$$\lambda x.\, e \longrightarrow \lambda x.\, e$$

$$\mathsf{raise}\ \epsilon \longrightarrow \mathsf{raise}\ \epsilon$$

$$\frac{e_1 \longrightarrow \mathsf{raise}\ \epsilon}{(e_1\ e_2) \longrightarrow \mathsf{raise}\ \epsilon}$$

$$\frac{e_1 \longrightarrow \lambda x.\, e' \qquad e_2 \longrightarrow \mathsf{raise}\ \epsilon}{(e_1\ e_2) \longrightarrow \mathsf{raise}\ \epsilon}$$

$$\frac{e_1 \longrightarrow \lambda x.\, e' \qquad e_2 \longrightarrow v_2 \qquad e'[x \mapsto v_2] \longrightarrow v}{(e_1\ e_2) \longrightarrow v} \quad v_2 \neq \mathsf{raise}\ \epsilon$$

$$\frac{e_0 \longrightarrow \mathsf{raise}\ \epsilon}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \mathsf{raise}\ \epsilon}$$

$$\frac{e_0 \longrightarrow \mathsf{true} \qquad e_1 \longrightarrow v_1}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow v_1}$$

$$\frac{e_0 \longrightarrow \mathsf{false} \qquad e_2 \longrightarrow v_2}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow v_2}$$

$$\frac{e_2 \longrightarrow \mathsf{raise}\ \epsilon \qquad e_1 \longrightarrow v}{\mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2 \longrightarrow v}$$

$$\frac{e_2 \longrightarrow v}{\mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2 \longrightarrow v} \quad v \neq \mathsf{raise}\ \epsilon$$

Table 2.1: Big-step semantics for exceptions.

$$\Gamma \vdash c : \tau_c$$

$$\Gamma \cup \{x : \tau\} \vdash x : \tau$$

$$\Gamma \vdash \textsf{raise}\ \epsilon : \tau$$

$$\frac{\Gamma \vdash e_1 : \textsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash (e_1\ e_2) : \tau}$$

$$\frac{\Gamma \cup \{x : \tau'\} \vdash e : \tau}{\Gamma \vdash \lambda x.\, e : \tau' \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textsf{handle}\ \epsilon\ \textsf{as}\ e_1\ \textsf{in}\ e_2 : \tau}$$

Table 2.2: Underlying type system rules.

We use the notation $e[x \mapsto e']$ to substitute a variable x for e' in an expression e. It will be the case that e' is a closed expression (i.e. without free variables) whenever we use substitutions so that we need not concern with the possibility of variable capture.

The objective of the exception analysis is to approximate what exceptions (if any) the evaluation of an expression may yield.

2.2.2 Underlying type system

The type and effect analysis will extend a standard type system with annotations. This underlying type system includes types of integers, booleans and functions:

$$\tau \quad ::= \quad \textsf{int} \ \mid\ \textsf{bool} \ \mid\ \tau_1 \to \tau_2$$

The typing relation is presented as judgements $\Gamma \vdash e : \tau$ where Γ is a set of assumptions for free variables (i.e. pairs $x : \tau$). We use τ_c for the type of a constant c (an integer or boolean). The typing rules are presented in Table 2.2.

2.2.3 Effects and annotated types

The key idea of type and effect systems is to annotate function types with an *effect* φ that delimits side-effects that may be triggered during the evaluation of the function. For the exception analysis, effects represent finite sets of exception tokens. We will, however, define effects inductively to facilitate the subsequent presentations of effect polymorphism and inference algorithms in Sections 2.2.7 and 2.2.8, respectively.

$$\varphi \quad ::= \quad \emptyset \;\mid\; \{\epsilon\} \;\mid\; \varphi_1 \cup \varphi_2 \tag{2.3}$$

Equality of effects is taken modulo axioms for commutativity, associativity and idempotency of $\cup$ and with null element $\emptyset$. We will also sometimes abuse notation and write effects as finite sets $\{\epsilon_1, \ldots, \epsilon_n\}$.

The syntax of annotated types is

$$\tau \quad ::= \quad \mathsf{int} \;\mid\; \mathsf{bool} \;\mid\; \tau_1 \xrightarrow{\varphi} \tau_2 \tag{2.4}$$

where int and bool are the types of constants and $\tau_1 \xrightarrow{\varphi} \tau_2$ is the type of a function that may raise exceptions *only* in φ. For example, the integer division operation can be given the annotated type $\mathsf{int} \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\{\mathsf{div0}\}} \mathsf{int}$ meaning that it might raise a division-by-zero exception.[1] By contrast, the addition operation can be given the type $\mathsf{int} \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\emptyset} \mathsf{int}$ manifesting that it cannot raise exceptions.

2.2.4 Subeffecting and subtyping

Effects can be ordered by a *subeffecting* relation $\subseteq$. Informally, $\varphi \subseteq \varphi'$ means that φ can be safely approximated by φ'. For the simple example of exception analysis, the subeffecting relation is just set-containment.

Subeffecting can be used to ensure that a type and effect system is a "conservative extension" of the underlying type system, i.e. to be able to derive an analysis for any expression that is typeable in the underlying type system. Consider, for example, the expression

$$\begin{aligned}\lambda y.\, \mathsf{if}\ y > 0\ \mathsf{then}\ (\lambda x.\, \mathsf{if}\ x > 0\ \mathsf{then\ raise\ pos\ else}\ x)\\ \mathsf{else}\ (\lambda x.\, \mathsf{if}\ x < 0\ \mathsf{then\ raise\ neg\ else}\ x)\end{aligned} \tag{2.5}$$

The two x-abstractions may raise distinct exceptions and so admit different annotated types:

$$(\lambda x.\, \mathsf{if}\ x > 0\ \mathsf{then\ raise\ pos\ else}\ x) \;:\; \mathsf{int} \xrightarrow{\{\mathsf{pos}\}} \mathsf{int}$$

$$(\lambda x.\, \mathsf{if}\ x < 0\ \mathsf{then\ raise\ neg\ else}\ x) \;:\; \mathsf{int} \xrightarrow{\{\mathsf{neg}\}} \mathsf{int}$$

[1] Note that currying allows distinguishing the effects of partial and full applications: the former cannot raise exceptions, while the later can.

Without subeffecting expression (2.5) would not admit an effect annotated type even though it is admits a type in the underlying type system. Subeffecting will allow us to “enlarge” the effects of both abstractions to

$$(\lambda x.\, \mathsf{if}\ x > 0\ \mathsf{then\ raise\ pos\ else}\ x) \ :\ \mathsf{int} \xrightarrow{\{\mathsf{neg},\, \mathsf{pos}\}} \mathsf{int}$$
$$(\lambda x.\, \mathsf{if}\ x < 0\ \mathsf{then\ raise\ neg\ else}\ x) \ :\ \mathsf{int} \xrightarrow{\{\mathsf{neg},\, \mathsf{pos}\}} \mathsf{int}$$

and obtain the type

$$\mathsf{int} \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\{\mathsf{neg},\, \mathsf{pos}\}} \mathsf{int}$$

for the whole expression (2.5).

Since types are annotated with effects, subeffecting induces a *subtyping* relation $\leqslant$ on annotated types. Informally, $\tau \leqslant \tau'$ means that τ can be safely approximated by τ'. The subtyping relation is formally defined by rules (2.6):

$$\tau \leqslant \tau \qquad\qquad \frac{\tau_1' \leqslant \tau_1 \qquad \tau_2 \leqslant \tau_2' \qquad \varphi \subseteq \varphi'}{\tau_1 \xrightarrow{\varphi} \tau_2 \leqslant \tau_1' \xrightarrow{\varphi'} \tau_2'} \tag{2.6}$$

Note that subtyping is *shape conformant* (or *structural*) i.e. if $\tau \leqslant \tau'$ then τ and τ' have the same underlying type but possibly distinct annotations; this is unlike more general kinds of subtyping that relate types with distinct constructors (e.g. a relation such as $\mathsf{int} \leqslant \mathsf{float}$ modelling coercion between numeric types).

Also note that the definition (2.6) is *covariant* on the right of the arrow but *contravariant* on the left, i.e. the subtyping order is reversed in the function domain. This is indeed the correct definition regardless of the precise semantics of side-effects; the intuition for this is that $A \xrightarrow{\varphi} B$ is interpreted as an implication $A \implies (B \wedge \varphi)$ and $\leqslant$ as logical consequence.

Finally, we remark that subeffecting alone is sufficient to ensure that the exception analysis is a conservative extension of the underlying type system; this is because the only type annotations are effects, unlike more complex type-based analysis (Amtoft et al. 1999, Reistad and Gifford 1994, Hughes and Pareto 1999). However, adding subtyping can still be useful to improve precision: while subeffecting requires enlarging effects at the point of definition, subtyping allows enlarging types at the points of use, thus limiting any precision loss to specific contexts.

2.2.5 Type and effect rules

The type and effect analysis is formulated in Table 2.3 as a set of typing rules that derive judgements with the form

$$\Gamma \vdash e : \tau \ \&\ \varphi$$

$$\Gamma \vdash c : \tau_c \mathbin{\&} \emptyset \tag{2.7}$$

$$\Gamma \cup \{x : \tau\} \vdash x : \tau \mathbin{\&} \emptyset \tag{2.8}$$

$$\Gamma \vdash \mathsf{raise}\ \epsilon : \tau \mathbin{\&} \{\epsilon\} \tag{2.9}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \mathbin{\&} \varphi_1 \qquad \Gamma \vdash e_2 : \tau \mathbin{\&} \varphi_2 \qquad \Gamma \vdash e_3 : \tau \mathbin{\&} \varphi_3}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \mathbin{\&} \varphi_1 \cup \varphi_2 \cup \varphi_3} \tag{2.10}$$

$$\frac{\Gamma \vdash e_1 : \tau' \xrightarrow{\varphi_0} \tau \mathbin{\&} \varphi_1 \qquad \Gamma \vdash e_2 : \tau' \mathbin{\&} \varphi_2}{\Gamma \vdash (e_1\ e_2) : \tau \mathbin{\&} \varphi_0 \cup \varphi_1 \cup \varphi_2} \tag{2.11}$$

$$\frac{\Gamma \cup \{x : \tau'\} \vdash e : \tau \mathbin{\&} \varphi}{\Gamma \vdash \lambda x.\, e : \tau' \xrightarrow{\varphi} \tau \mathbin{\&} \emptyset} \tag{2.12}$$

$$\frac{\Gamma \vdash e_1 : \tau \mathbin{\&} \varphi_1 \qquad \Gamma \vdash e_2 : \tau \mathbin{\&} \varphi_2}{\Gamma \vdash \mathsf{handle}\ \epsilon\ \mathsf{as}\ e_1\ \mathsf{in}\ e_2 : \tau \mathbin{\&} \varphi_1 \cup (\varphi_2 \setminus \{\epsilon\})} \tag{2.13}$$

$$\frac{\Gamma \vdash e : \tau \mathbin{\&} \varphi}{\Gamma \vdash e : \tau \mathbin{\&} \varphi'} \quad \text{if } \varphi \subseteq \varphi' \tag{2.14}$$

Table 2.3: Type and effect rules for exception analysis.

where Γ is a set of type assumptions for free identifiers, e is an expression, τ is an annotated type and φ is the effect associated with e. We describe each rule informally:

- Rules (2.7) and (2.8) specify the type and an empty effect $\emptyset$ for constants and identifiers: under a call-by-value semantics evaluation of these expressions cannot raise exceptions.
- Rule (2.9) specifies the most precise effect for an explicit raise, namely, the singleton exception raised; note that the result type τ is arbitrary.
- Rule (2.10) overestimates the effect of a conditional as the union of the effects of all sub-expressions.
- Rule (2.11) specifies the combination of effects for an application: the result is a union of the effects φ_1, φ_2 for evaluating both sub-expressions plus the "latent effect" φ_0 for the function itself.
- Conversely, rule (2.12) transposes the effect φ of an expression e into the annotation in the arrow type for the abstraction $\lambda x.\, e$. The evaluation the lambda-abstraction has an empty effect; this is because the effects of the function are delayed until the point of application.
- Rule (2.13) specifies the type and effect of the handle construct: an exception ϵ raised in e_2 is caught and therefore the result effect masks it using an "effect-difference" operation $\varphi_2 \setminus \{\epsilon\}$ (the definition is straightforward and we omit it). Note that exceptions other than ϵ raised in e_2 or those raised by e_1 are propagated to the outer scope.
- Rule (2.14) allows subeffecting only; it would be possible to allow subtyping as well (as in Nielson et al. 1999):

  $$\frac{\Gamma \vdash e : \tau \And \varphi}{\Gamma \vdash e : \tau' \And \varphi'} \qquad \text{if } \tau \leqslant \tau' \text{ and } \varphi \subseteq \varphi'$$

  We do not consider the extended rule here to avoid the treatment of subtyping in the inference algorithm. In any case, subtyping would only improve precision; subeffecting alone is sufficient to ensure that the exception analysis is a conservative extension of the underlying type system.

2.2.6 Semantic correctness

The correctness of the type and effect analysis can be formulated as a "subject reduction" property: if a type and effect can is inferred for an expression, it is also admissible for the result of evaluation.

For the particular exception analysis, this is formulated in the following theorem.

Theorem 2.1 *If* $\emptyset \vdash e : \tau \ \& \ \varphi$ *and* $e \longrightarrow v$*, then* $\emptyset \vdash v : \tau \ \& \ \varphi$*.*

In particular, if $\emptyset \vdash e : \tau \ \& \ \varphi$ and $e \longrightarrow \mathsf{raise}\ \epsilon$, then applying the above theorem we obtain $\emptyset \vdash \mathsf{raise}\ \epsilon : \tau \ \& \ \varphi$. By inspection of Table 2.3 we can see that the only type rules that could be applied to raise are (2.9) and (2.14). We conclude that $\{\epsilon\} \subseteq \varphi$, i.e. the analysis obtains an upper-approximation of the exceptions raised.

The proof of Theorem 2.1 is by induction on the big-step reduction $e \longrightarrow v$ together with a standard "substitution lemma" to allow replacing variables with expressions of the correct type. We omit the proof which is similar to the one presented in Nielson et al. (1999, pages 295–297).

2.2.7 Type and effect polymorphism

For simplicity the language and type rules considered so far did not include polymorphic definitions. We will now add let-bound polymorphism by extending terms with an expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ and a type rule that allows quantified types for x in e_2. Moreover, it is possible to use polymorphism to obtain a more precise analysis by quantifying over effects as well as types. The extended syntax of terms and types is

$$\begin{array}{lcl}
e & ::= & \cdots \ \mid \ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
\varphi & ::= & \beta \ \mid \ \emptyset \ \mid \ \{\epsilon\} \ \mid \ \varphi_1 \cup \varphi_2 \\
\tau & ::= & \alpha \ \mid \ \mathsf{int} \ \mid \ \mathsf{bool} \ \mid \ \tau_1 \xrightarrow{\varphi} \tau_2 \\
\sigma & ::= & \tau \ \mid \ \forall \gamma.\, \sigma \\
\gamma & ::= & \alpha \ \mid \ \beta \\
\alpha & ::= & \mathsf{a} \ \mid \ \mathsf{b} \ \mid \ \mathsf{c} \ \mid \ \ldots \\
\beta & ::= & '0 \ \mid \ '1 \ \mid \ '2 \ \mid \ \ldots
\end{array}$$

where α are *type variables*, β are *effect variables* and σ are *quantified types* (i.e. type schemes).

Table 2.4 lists the new type rules for the let-expression 2.15 and for introduction and elimination of type quantifiers (2.16), (2.17), (2.18); the rules of Table 2.3 remain unchanged.

$$\frac{\Gamma \vdash e_1 : \sigma_1 \ \& \ \varphi_1 \qquad \Gamma \cup \{x : \sigma_1\} \vdash e_2 : \tau_2 \ \& \ \varphi_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \ \& \ \varphi_1 \cup \varphi_2} \tag{2.15}$$

$$\frac{\Gamma \vdash e : \sigma \ \& \ \varphi}{\Gamma \vdash e : \forall \gamma.\, \sigma \ \& \ \varphi} \quad \text{if } \gamma \text{ does not occur free in } \Gamma \text{ and } \varphi \tag{2.16}$$

$$\frac{\Gamma \vdash e : \forall \alpha.\, \sigma \ \& \ \varphi}{\Gamma \vdash e : \sigma[\alpha \mapsto \tau'] \ \& \ \varphi} \tag{2.17}$$

$$\frac{\Gamma \vdash e : \forall \beta.\, \sigma \ \& \ \varphi}{\Gamma \vdash e : \sigma[\beta \mapsto \varphi'] \ \& \ \varphi} \tag{2.18}$$

Table 2.4: Extensions for type and effect polymorphism.

Example 2.2 Consider a program that starts with the definition of a higher-order composition function,

$$\mathsf{let}\ \mathit{compose} = \lambda f.\, \lambda g.\, \lambda x.\, f\ (g\ x)\ \mathsf{in}\ e$$

where the expression e is the remaining part of the program. Using type and effect polymorphism, we can derive a quantified type for *compose*

$$\forall \mathsf{a}\, \forall \mathsf{b}\, \forall \mathsf{c}\, \forall '0\, \forall '1.\, (\mathsf{b} \xrightarrow{'0} \mathsf{c}) \xrightarrow{\emptyset} (\mathsf{a} \xrightarrow{'1} \mathsf{b}) \xrightarrow{\emptyset} \mathsf{a} \xrightarrow{'0 \cup '1} \mathsf{c}$$

which specifies the "most general" annotated type with instances such as

$$(\mathsf{int} \xrightarrow{\emptyset} \mathsf{int}) \xrightarrow{\emptyset} (\mathsf{int} \xrightarrow{\emptyset} \mathsf{int}) \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\emptyset} \mathsf{int}$$

and

$$(\mathsf{int} \xrightarrow{\{\mathsf{neg}\}} \mathsf{bool}) \xrightarrow{\emptyset} (\mathsf{int} \xrightarrow{\{\mathsf{pos}\}} \mathsf{int}) \xrightarrow{\emptyset} \mathsf{int} \xrightarrow{\{\mathsf{neg},\, \mathsf{neg}\}} \mathsf{bool}\ .$$

Note that with type but not effect polymorphism we would only be able to derive a type for *compose* annotated with the effects of *all* uses; this would lead to an over-estimation where each use shares the effects of all others.

By contrast, quantification on effects allows the analysis of polymorphic functions to be *polyvariant*: each application is given a range of effects specific to its arguments; this is particularly important for generic functions such as higher-order combinators that are typically used in very different contexts.

Note also that effect polymorphism allows expressing the analysis of *compose* without advance knowledge of its uses. This means that the analysis is *compositional*: it does require that the whole program be present and can be applied to separately-compiled modules or libraries. □

We remark that the addition of polymorphism to the exception analysis is uncharacteristically simple. In general, the combination of polymorphism and imperative side-effects (e.g. references) requires restrictions on the use of the generalisation rule to retain soundness of type inference (Tofte 1988, Wright 1995).

2.2.8 Inference algorithms

Tables 2.3 and 2.4 present the type and effect analysis as proof systems that require guessing suitable types for sub-expressions; to obtain an automatic analysis we need an algorithm for *type and effect reconstruction*.

A first problem is the presence of non-structural type rules such as those for subeffecting (2.14), subtyping, generalisation (2.16) and instantiation (2.17), (2.18): these rules can occur in arbitrary points of a derivation and therefore some "canonical" choice has to be made; this is usually done by establishing a *proof normalisation* result, i.e. that uses of non-structural rules can be restricted to specific syntax points without incurring a loss of typeability.

Let-bound polymorphism is a suitable choice for normalising the uses of generalisation and instantiation: generalise all suitable type and effect variables of let-bound identifiers[2] and instantiate all quantified variables just after the use of a variable. Subeffecting can be normalised by allowing over-approximation of effects in all rules, i.e. by adding an arbitrary effect $\ldots \cup \varphi'$ to the conclusions of (2.7), (2.8), (2.9) and (2.12).

Type inference for type and effect systems with subeffecting but not subtyping can be implemented as an extension of the well-known algorithm W of Damas (1985). The key insight is to restrict types to the subset of *simple types* $\widehat{\tau}$ whose annotations must be variables:

$$\widehat{\tau} \quad ::= \quad \alpha \ \mid \ \mathsf{int} \ \mid \ \mathsf{bool} \ \mid \ \widehat{\tau}_1 \xrightarrow{\beta} \widehat{\tau}_2$$

To allow expressing complex effects (i.e. non-variables) the algorithm collects separate *lower-bound constraints* C over effect variables:

$$\begin{aligned} C \quad &::= \quad \emptyset \ \mid \ \{\beta \supseteq \varphi\} \ \mid \ C_1 \cup C_2 \\ \varphi \quad &::= \quad \emptyset \ \mid \ \beta \ \mid \ \{\epsilon\} \ \mid \ \varphi_1 \cup \varphi_2 \end{aligned}$$

The reason for restricting the algorithm to simple types is that these form a free algebra in which equality constraints can be solved by first-order unification (Robinson 1971) just as in ordinary Damas-Milner type inference. By contrast, the algebra

[2] That is, those that do not occur free in the type assumption or effect.

$$\begin{aligned}
\mathcal{U}(\mathsf{int},\, \mathsf{int}) &= id \\
\mathcal{U}(\mathsf{bool},\, \mathsf{bool}) &= id \\
\mathcal{U}(\widehat{\tau}_1 \xrightarrow{\beta} \widehat{\tau}_2,\, \widehat{\tau}_1' \xrightarrow{\beta'} \widehat{\tau}_2') &= \begin{array}{ll} \text{let} & \theta_0 = [\beta \mapsto \beta'] \\ & \theta_1 = \mathcal{U}(\theta_0 \widehat{\tau}_1,\, \theta_0 \widehat{\tau}_1') \\ & \theta_2 = \mathcal{U}(\theta_1 \theta_0 \widehat{\tau}_2,\, \theta_1 \theta_0 \widehat{\tau}_2') \\ \text{in} & \theta_2 \circ \theta_1 \circ \theta_0 \end{array} \\
\mathcal{U}(\alpha,\, \widehat{\tau}) = \mathcal{U}(\widehat{\tau},\, \alpha) &= \begin{cases} [\alpha \mapsto \widehat{\tau}] & \text{if } \alpha \text{ does not occur in } \widehat{\tau} \\ \text{fails} & \text{otherwise} \end{cases} \\
\mathcal{U}(\widehat{\tau},\, \widehat{\tau}') & \quad \text{fails in all other cases}
\end{aligned}$$

Table 2.5: Unification of simple types.

of effects is non-free (e.g. $\cup$ is associative, commutative and has a empty element $\emptyset$). By segregating effects to separate constraints, it becomes possible to use the simple unification to solve type equalities and deal with the non-free algebra of effects in a separate constraint solver.

Table 2.6 presents an excerpt of the reconstruction algorithm as judgements

$$\widehat{\Gamma} \vdash_{\mathsf{RA}} e : (\widehat{\tau},\, \varphi,\, C,\, \theta)$$

where $\widehat{\Gamma}$ is a set of (simple) type assumptions, e is an expression and the output is a 4-tuple of: a simple type $\widehat{\tau}$, an effect φ, a set of lower-bound constraints C and a substitution θ. For simplicity, we include only the rules for constants, abstraction and application; the omitted cases (conditionals and exception handling) are straightforward but tedious.

The main difference between the proof systems of Table 2.3 and Table 2.6 is that the latter does not require guessing types of sub-expressions; instead, it uses "fresh" variables for both types and effects and uses unification to impose equality constraints between (simple) types.

The unification algorithm $\mathcal{U}$ in Table 2.5 takes two simple types $\widehat{\tau}$, $\widehat{\tau}'$ and yields the "smallest" substitution θ such that $\theta\widehat{\tau} \equiv \theta\widehat{\tau}'$ (or fails, if no such substitution exists). Note that substitutions bind both type and effect variables and therefore are applied to types, effects and constraints.

Each rule of Table 2.6 is applicable to a single expression syntax node; thus, the rules can be read as an algorithm for reconstructing the type and effect of an

$$\widehat{\Gamma} \vdash_{\mathsf{RA}} c : (\tau_c,\, \emptyset,\, \emptyset,\, id)$$

$$\widehat{\Gamma} \cup \{x : \widehat{\tau}\} \vdash_{\mathsf{RA}} x : (\widehat{\tau},\, \emptyset,\, \emptyset, id)$$

$$\widehat{\Gamma} \vdash_{\mathsf{RA}} \mathsf{raise}\ \epsilon : (\alpha,\, \{\epsilon\},\, \emptyset, id)$$

$$\frac{\widehat{\Gamma} \cup \{x : \alpha\} \vdash_{\mathsf{RA}} e : (\widehat{\tau},\, \varphi,\, C,\, \theta)}{\widehat{\Gamma} \vdash_{\mathsf{RA}} \lambda x.\, e : (\theta\alpha \xrightarrow{\beta} \widehat{\tau},\, \emptyset,\, \{\beta \supseteq \varphi\} \cup C,\, \theta)} \quad \alpha,\, \beta \text{ are fresh variables}$$

$$\frac{\begin{array}{l}\widehat{\Gamma} \vdash_{\mathsf{RA}} e_1 : (\widehat{\tau}_1,\, \varphi_1,\, C_1,\, \theta_1) \qquad \theta_1\widehat{\Gamma} \vdash_{\mathsf{RA}} e_2 : (\widehat{\tau}_2,\, \varphi_2,\, C_2,\, \theta_2) \\ \theta_3 = \mathcal{U}(\widehat{\tau}_2 \xrightarrow{\beta} \alpha,\, \theta_2\widehat{\tau}_1)\end{array}}{\widehat{\Gamma} \vdash_{\mathsf{RA}} (e_1\ e_2) : (\theta_3\alpha,\, \theta_3\theta_2\varphi_1 \cup \theta_3\varphi_2 \cup \{\theta_3\beta\},\, \theta_3\theta_2C_1 \cup \theta_3C_2,\, \theta_3 \circ \theta_2 \circ \theta_1)}$$

Table 2.6: Algorithmic typing judgements for exception analysis (excerpt).

expression.

Extending the inference algorithm with let-bound polymorphism is straightforward: quantification of variables is handled at the let and instantiation is handled at the use of variables by introducing fresh type and effect variables. The type and effect system for region inference of Talpin and Jouvelot (1992) combines polymorphism and effects (but not subtyping).

Type reconstruction algorithms for subtyping usually require extending the proof system with explicit type inequality constraints (Mitchell 1984, Fuh and Mishra 1988); this is needed to obtain syntactic completeness, i.e. an algorithm that computes a principal solution from which any valid typing can be derived. This approach is followed in Nielson et al. (1996*a*,*b*) although completeness of the algorithm is left as an open problem. For shape conformant subtyping typical of type and effect systems it is possible to employ a simpler two-stage approach: first the underlying types are inferred and then the subtyping inequalities are translated to constraints on the annotations (Reistad and Gifford 1994); such an algorithm will not be complete, i.e. it may compute a type and effect that is not minimal.

2.2.9 Concluding remarks

The type and effect discipline has some strengths compared to the other main approaches for program analysis (e.g. based on data-flow or abstract interpretation): it deals naturally with higher-order functions by means of annotations on arrow types;

it allows separate analysis of modules by communicating information via extended type signatures; the latter also provide a natural mechanism for reporting the results of analysis to the user.

Type and effect systems do not naturally fit languages with non-strict semantics, e.g. lazy evaluation; this can be witnessed in the type rule (2.11) for application:

$$\frac{\Gamma \vdash e_1 : \tau' \xrightarrow{\varphi_0} \tau \ \& \ \varphi_1 \qquad \Gamma \vdash e_2 : \tau' \ \& \ \varphi_2}{\Gamma \vdash (e_1 \ e_2) : \tau \ \& \ \varphi_0 \cup \varphi_1 \cup \varphi_2}$$

The effect $\varphi_0 \cup \varphi_1 \cup \varphi_2$ of the application subsumes the effect φ_2 of the argument, thus modelling a strict application.

Effect systems where initially proposed to address the problems of combining ML-style polymorphism with implicit side-effects. But *implicit* effects are not useful in a non-strict language because the order of evaluation is dependent on demand. In lazy functional languages the sequentiality of effects must be controlled explicitly, e.g. using *monads* (Wadler 1993, Benton et al. 2000).

However, the monadic and effect-system approaches are not completely apart: Wadler (1998*a*) has shown that monads can be parameterised by effects and that the inference rules and algorithms of type and effect systems carry over to the monadic translation.

2.3 Abstract interpretation

Abstract interpretation (Cousot and Cousot 1977, 1992*a*) is a framework for program analysis based on approximating computations on *concrete values* by computations on *abstract properties* of values. The key idea is to approximate the concrete domain of values by a more coarse domain of abstract properties and lift the concrete operations to sound abstract approximations. The static analysis is constructed by interpreting the program in the non-standard abstract domain.

2.3.1 Concrete and abstract domains

The theory of abstract interpretation is concerned with establishing sound approximations independently of the syntactical characteristics of the programming language (or, more generally, model of computation). It is therefore usual to start by defining sets $\mathcal{P}^\natural$ of *concrete properties* and $\mathcal{P}^\sharp$ of *abstract properties.*

The concrete properties are derived from some semantic description of the language (e.g. they can be sets of values, states or given by a “collecting” semantics);

the requirement for $\mathcal{P}^\natural$ is that is a complete proof method for the intended properties under analysis; therefore elements of $\mathcal{P}^\natural$ are usually not machine representable. Conversely, the abstract properties $\mathcal{P}^\sharp$ will typically be both finitely representable and computable.

The sets of concrete and abstract properties are then instrumented with partial orders representing the relative precision of descriptions; a common scenario is to consider properties to be lattices ($\mathcal{P}$, $\sqsubseteq$, $\bot$, $\top$, $\sqcup$, $\sqcap$). The convention is that precision is lost when moving upwards in the lattice: if p, $p' \in \mathcal{P}$ satisfy $p \sqsubseteq p'$ then any value described by p is also described by p', i.e. p entails p'. The bottom element $\bot$ represents the most precise property (i.e. divergent or non-reachable computations); the top element $\top$ represent the least precise property (absence of information). Note that it is possible for properties to be incomparable, i.e. when neither $p \sqsubseteq p'$ nor $p' \sqsubseteq p$ holds.

2.3.2 Correspondence between concrete and abstract properties

The correspondence between concrete and abstract properties $c \in \mathcal{P}^\natural$ and $a \in \mathcal{P}^\sharp$ can be established in many ways; one of the most common scenarios is to require the existence of two monotone functions $\alpha : \mathcal{P}^\natural \to \mathcal{P}^\sharp$ and $\gamma : \mathcal{P}^\sharp \to \mathcal{P}^\natural$ called *abstraction* and *concretisation*, respectively. Informally, $\alpha(c)$ should be the "smallest" abstract representative of a concrete property c; dually, $\gamma(a)$ should be the "largest" concrete property described by an abstraction a.

The classical *Galois connection* framework requires that the abstraction and concretisation functions satisfy (2.19) and (2.20):

$$\forall c \in \mathcal{P}^\natural \quad c \sqsubseteq^\natural \gamma(\alpha(c)) \tag{2.19}$$

$$\forall a \in \mathcal{P}^\sharp \quad \alpha(\gamma(a)) \sqsubseteq^\sharp a \tag{2.20}$$

Condition (2.19) states that we may loose precision moving from concrete to abstract lattice and back again (though we do not loose safety); condition (2.20) states that we do not loose precision in the inverse direction.

In the presence of a Galois connection we can state the soundness relation between a concrete and abstract property using either the concrete or abstract orders. More precisely, $a \in \mathcal{P}^\sharp$ is a sound approximation of $c \in \mathcal{P}^\natural$ if either (2.21) or (2.22) holds:

$$c \sqsubseteq^\natural \gamma(a) \tag{2.21}$$

$$\alpha(c) \sqsubseteq^\sharp a \tag{2.22}$$

However, we shall consider the slightly more general setting where not every concrete value has a “best” abstraction, e.g. when the domain of abstract properties is an incomplete lattice (Cousot and Halbwachs 1978, Cousot and Cousot 1992*b*). In that situation we can do without the abstraction function and define the soundness relation using condition (2.21).

2.3.3 Approximation of fixed points

We now introduce a further assumption that the lattice $\mathcal{P}^\natural$ of concrete properties is *complete* so that we can define the invariant properties of recursive or iterative computations as fixed points.

Assume then that the semantics of a “basic block” of computation (e.g. a state transition or the body of a loop or recursive computation) is given by a continuous function $f : \mathcal{P}^\natural \to \mathcal{P}^\natural$ on concrete properties. The semantics of an iterative computation can then be obtained as the limit of an ascending chain of concrete iterates $\{(c_n)_{n\in\mathbb{N}}\}$:

$$c_0 \overset{\text{def}}{=} \bot^\natural \qquad c_{n+1} \overset{\text{def}}{=} f(c_n) \tag{2.23}$$

By completeness of $\mathcal{P}^\natural$ the limit $\bigsqcup\{(c_n)_{n\in\mathbb{N}}\}$ exists; by continuity of f we have $\bigsqcup\{(c_n)_{n\in\mathbb{N}}\} = \bigsqcup_{n\geq 0} f^n(\bot^\natural) = \mathit{fix}(f)$.

It will usually be the case that the least fixed point $\mathit{fix}(f)$ is incomputable and so we are interested in obtaining a computable approximation using a monotone abstract semantics function $f^\sharp : \mathcal{P}^\sharp \to \mathcal{P}^\sharp$ describing a transition in abstract properties. Note that the requirement for monotonicity is quite natural for program analysis: it merely amounts to saying that $f^\sharp$ cannot yield more precise results from less precise initial approximations. Note also that we do not assume that the lattice $\mathcal{P}^\sharp$ is complete nor that $f^\sharp$ is continuous.

The soundness relation between f and $f^\sharp$ can specified using a Galois connection, i.e. abstraction and concretisation functions. In that case $f^\sharp$ is completely determined by f, α and γ:

$$f^\sharp = \alpha \circ f \circ \gamma \tag{2.24}$$

Informally, condition (2.24) says that computing with $f^\sharp$ should yield the same result as first using γ to project into the concrete domain, computing with f and then using α to get back to the abstract domain.

In the absence of a Galois connection, it is possible to employ the concretisation function alone to specify the soundness relation using (2.21) in a pointwise manner:

$$\forall a \in \mathcal{P}^\sharp \quad f(\gamma(a)) \sqsubseteq^\natural \gamma(f^\sharp(a)) \tag{2.25}$$

Condition (2.25) says that computing with $f^\sharp$ and then projecting into the concrete domain must yield a upper-approximation of first projecting into the concrete domain and then computing with f.

Consider now the sequence of abstract iterates $\{(a_n)_{n\in\mathbb{N}}\}$ generated by $f^\sharp$:

$$a_0 \stackrel{\text{def}}{=} \bot^\sharp \qquad a_{n+1} \stackrel{\text{def}}{=} f^\sharp(a_n) \tag{2.26}$$

It is immediate that $\{(a_n)_{n\in\mathbb{N}}\}$ is an ascending chain in $\mathcal{P}^\sharp$ by the monotonicity of $f^\sharp$. In the simple situation where the abstract lattice of properties $\mathcal{P}^\sharp$ is *finite* the iteration (2.26) will converge to the least fixed point in a finite number of iterations.

However, this is not the case for lattices that capture numerical properties, e.g. the lattices of integer intervals or convex polyhedra (Cousot and Halbwachs 1978). If the abstract lattice is incomplete or does not satisfy the finite ascending chain condition (see Appendix A.1.8) then the limit of (2.26) might not exist or the iteration might not converge finitely; in either case the abstract iteration does not give an effective computational method for obtaining a sound abstraction.

The solution is to replace (2.26) by another iteration that is an upper bound of the original and does stabilise in a finite number of steps; this can be done using a *widening operator*.

Widening operators

When the lattice of abstract values does not satisfy the ascending chain condition, we need some heuristic for extrapolating fixed point iterations. This is designated a *widening operator*.[3]

Definition 2.3 *A total function $\nabla : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ on a partially ordered set $(\mathcal{P}, \sqsubseteq)$ is a* widening operator *if and only if:*

1. *For all $x, y \in \mathcal{P}$ we have $x \sqsubseteq x\nabla y$ and $y \sqsubseteq x\nabla y$;*
2. *If for all n, $x_n \in \mathcal{P}$ and $x_n \sqsubseteq x_{n+1}$, then the sequence $y_n \in \mathcal{P}$ defined by*

$$y_0 \stackrel{\mathit{def}}{=} x_0 \qquad y_{n+1} \stackrel{\mathit{def}}{=} y_n \nabla x_{n+1}$$

eventually stabilises, i.e. there exists $k \geq 0$ such that for all n, $n \geq k$ implies $y_n = y_k$.

[3] There is some flexibility in the definition of widening operators; we use the formulation of (Nielson et al. 1999).

The first condition of this definition requires that ∇ is an upper bound operator (though not necessarily the *least* upper bound); the second condition requires that ∇ transforms ascending iterations into finitely stabilising iterations.

Assume we have a widening operator ∇ for the lattice of abstract properties $\mathcal{P}^\sharp$; then the following iteration

$$\begin{aligned} a_0 &= \bot^\sharp \\ a_{n+1} &= a_n && \text{if } f^\sharp(a_n) \sqsubseteq^\sharp a_n \\ a_{n+1} &= a_n \nabla f^\sharp(a_n) && \text{otherwise} \end{aligned} \tag{2.27}$$

eventually stabilises (Nielson et al. 1999, pages 227–228). Therefore, we can compute successive iterates a_0, a_1, ... until the condition $f^\sharp(a_k) \sqsubseteq^\sharp a_k$ is satisfied; by the properties of the widening, this is guaranteed to happen in a finite number of iterations; then

$$\begin{aligned} & f^\sharp(a_k) \sqsubseteq^\sharp a_k && \text{by the termination condition} \\ \Longrightarrow\ & \gamma(f^\sharp(a_k)) \sqsubseteq^\natural \gamma(a_k) && \text{by monotonicity of } \gamma \\ \Longrightarrow\ & f(\gamma(a_k)) \sqsubseteq^\natural \gamma(a_k) && \text{by hypothesis (2.25)} \\ \Longrightarrow\ & \mathit{fix}(f) \sqsubseteq^\natural \gamma(a_k) && \text{by the fixed point theorem } A.1 \end{aligned}$$

The last line says that the abstract property a_k obtained from (2.27) is a sound approximation of $\mathit{fix}(f)$ as required.

2.3.4 Abstract interpretation of numerical properties

Several applications of analysis and verification require approximating the dynamic values of program variables, e.g. the elimination of array bounds check in an optimising compiler. It is sometimes desirable to approximate not just individual values, but also *relations* between them, e.g. the input-output relation between list lengths of the size analysis example in Chapter 1.

In general, such analyses construct finite representations of (possibly infinite) sets of vectors in an n-dimensional space; each variable or parameter of the analysed program is associated with one particular dimension. Thus, the concrete domain for numerical relations between n variables is $\wp(\mathbb{Z}^n)$.

It is immediate that this domain is not machine representable, so we will be interested in computable sound approximations (i.e. upper bounds in the set containment order). Two of the earliest but still most relevant abstract domains for this purpose are the *intervals of integers* (Cousot and Cousot 1976) and *convex polyhedra* (Cousot and Halbwachs 1978). Unlike elements of $\wp(\mathbb{Z}^n)$, both intervals and polyhedra are machine representable and support a rich algebra of computable operations.

2.3.5 Lattice of intervals

We start by a simple case, namely, approximating the range of values of a single integer variable. A concrete element is then $X \in \wp(\mathbb{Z})$, that is, a set $X \subseteq \mathbb{Z}$ of integer values. To approximate this we can use an *interval* that contains X. This can be represented by a pair $[l,\, r]$ of bounds, which are either integers, $-\infty$ or $+\infty$; a special symbol $\perp$ represents the empty interval. Formally, we define the integer intervals by

$$\mathbf{Interval} = \{\perp\} \cup \{\, [l,\, r] \;:\; l \in \mathbb{Z} \cup \{-\infty\},\, r \in \mathbb{Z} \cup \{+\infty\},\, l \leq r\,\}$$

where the order on the integers is extended to $-\infty$ and $+\infty$ by $-\infty \leq x$, $x \leq +\infty$ and $-\infty \leq +\infty$ for all $x \in \mathbb{Z}$.

The partial order $\sqsubseteq$ of *interval containment* is defined by

$$\mathit{int}_1 \sqsubseteq \mathit{int}_2 \overset{\text{def}}{\iff} \inf \mathit{int}_2 \leq \inf \mathit{int}_1 \wedge \sup \mathit{int}_1 \leq \sup \mathit{int}_2$$

where

$$\inf \perp = +\infty \quad \sup \perp = -\infty$$
$$\inf\, [l,\, r] = l \quad \sup\, [l,\, r] = r\,.$$

Then ($\mathbf{Interval}$, $\sqsubseteq$, $\perp$, $[-\infty,\, +\infty]$, $\sqcup$, $\sqcap$) is a complete lattice (Nielson et al. 1999, pages 221–222), where the join $\sqcup$ and meet $\sqcap$ are defined by

$$\begin{aligned}
[l,\, r] \sqcup [l',\, r'] &\overset{\text{def}}{=} [\min(l, l'),\, \max(r, r')] \\
\perp \sqcup \mathit{int} &\overset{\text{def}}{=} \mathit{int} \sqcup \perp \overset{\text{def}}{=} \mathit{int} \\
[l,\, r] \sqcap [l',\, r'] &\overset{\text{def}}{=} \begin{cases} [\max(l, l'),\, \min(r, r')]\,, & \text{if } \max(l, l') \leq \min(r, r') \\ \perp & \text{otherwise} \end{cases} \\
\perp \sqcap \mathit{int} &\overset{\text{def}}{=} \mathit{int} \sqcap \perp \overset{\text{def}}{=} \perp
\end{aligned}$$

and min and max extend to $-\infty$ and $+\infty$ in the natural way.

Operations on intervals

The best approximation of a non-empty set $X \subseteq \mathbb{Z}$ of integers is the interval $[\inf X,\, \sup X]$; dually, an interval $[l,\, r]$ represents the set $X = \{x \in \mathbb{Z} : l \leq x \leq r\}$; the interval $\perp$ represents the empty set. More generally, we define the abstraction and concretisation functions as follows:

$$\begin{array}{rcll}
\alpha & : & \wp(\mathbb{Z}) \to \mathbf{Interval} & \\
\alpha(\emptyset) & \overset{\text{def}}{=} & \perp & \\
\alpha(X) & \overset{\text{def}}{=} & [\inf X,\, \sup X] & (X \neq \emptyset)
\end{array}
\qquad
\begin{array}{rcl}
\gamma & : & \mathbf{Interval} \to \wp(\mathbb{Z}) \\
\gamma(\perp) & \overset{\text{def}}{=} & \emptyset \\
\gamma([l,\, r]) & \overset{\text{def}}{=} & \{x \in \mathbb{Z} : l \leq x \leq r\}
\end{array}$$

It is straightforward to verify that (α, γ) form a Galois connection. This means we can "lift" operations from integer to intervals in a pointwise-manner; for example, the addition operation can be approximated by:

$$\begin{gathered}\perp + \mathit{int} = \mathit{int} + \perp = \perp \\ [l,\, r] + [l',\, r'] = [l + l',\, r + r']\end{gathered}$$

Similar approximations can be derived for other arithmetic operations.

Widening operators for intervals

We are now almost in condition to use the lattice of interval to approximate ranges of program variables. However, because the lattice of intervals has infinite height, we have no termination guarantee for iterative approximation of fixed points. This would prevent the applicability of abstract interpretation to iterative or recursive programs. Fortunately, we can resort to the general method of Section 2.3.3 for ensuring finite convergence of iterations, namely, employing a *widening operator*.

Widening operators for intervals go back to one of the first publications on abstract interpretation; the widening operator proposed in Cousot and Cousot (1976) simply extrapolates bounds that are not stable to $+\infty$ or $-\infty$. The formal definition is as follows:

$$\begin{aligned} \perp \nabla \mathit{int} &\stackrel{\text{def}}{=} \mathit{int} \nabla \perp \stackrel{\text{def}}{=} \mathit{int} \\ [z_1,\, z_2] \nabla [z_1',\, z_2'] &\stackrel{\text{def}}{=} [l,\, r] \\ \text{where } l &= \begin{cases} z_1 & \text{if } z_1 \leq z_1' \\ -\infty & \text{otherwise} \end{cases} \\ r &= \begin{cases} z_2 & \text{if } z_2' \leq z_2 \\ +\infty & \text{otherwise} \end{cases} \end{aligned}$$

Example 2.4 Consider the simple imperative program with assignments and an input instruction that yields a boolean result:

```
$i := 0$; $c :=$ true
while $c$ do
    $i := i + 2$
    read($c$)
```
(2.28)

We wish to approximate the values of the variable i inside the loop (2.28) using abstract interpretation on the lattice of intervals. For this simple example, it is easy to verify that the function specifying the concrete transition associated with one iteration of the loop is

$$f : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$$
$$f(X) = \{0\} \cup \{x + 2 : x \in X\}$$

and the concrete semantics is $\mathit{fix}(f) = \bigcup_{n \geq 0} f^n(\emptyset) = \{0,\ 2,\ 4, \ldots\} = 2\mathbb{N}$, i.e. the variable i ranges over positive even numbers.

We can now specify the abstract transition function using intervals rather than sets:

$$f^\sharp : \textbf{Interval} \to \textbf{Interval}$$
$$f^\sharp(\mathit{int}) = [0,\ 0] \sqcup (\mathit{int} + [2,\ 2])$$

where $+$ is the addition of intervals. Computing the sequence of abstract iterates, defined by $a_0 = \bot$ and $a_{n+1} = f^\sharp(a_n)$, yeilds:

$$\begin{aligned} a_0 &= \bot \\ a_1 &= f^\sharp(a_0) = [0,\ 0] + (\bot + [2,\ 2]) = [0,\ 0] \\ a_2 &= f^\sharp(a_1) = [0,\ 0] + ([0,\ 0] + [2,\ 2]) = [0,\ 2] \\ a_3 &= f^\sharp(a_2) = [0,\ 0] + ([0,\ 2] + [2,\ 2]) = [0,\ 4] \\ &\ \vdots \end{aligned}$$

It should be apparent that the limit $\bigsqcup_{n \geq 0} a_n = \bigsqcup\{\bot,\ [0,\ 0],\ [0,\ 2],\ [0,\ 4] \ldots\} = [0,\ +\infty]$ will *not* be obtained in a finite number of iterations.

To ensure finite convergence, we now consider the abstract iteration with widening, defined by $a'_0 \stackrel{\text{def}}{=} \bot$ and $a'_{n+1} \stackrel{\text{def}}{=} a'_n \nabla f^\sharp(a'_n)$:

$$\begin{aligned} a'_0 &= \bot \\ a'_1 &= \bot \nabla f^\sharp(a'_0) = \bot \nabla [0,\ 0] = [0,\ 0] \\ a'_2 &= [0,\ 0] \nabla f^\sharp(a'_1) = [0,\ 0] \nabla [0,\ 2] = [0,\ +\infty] \\ a'_3 &= [0,\ 0] \nabla f^\sharp(a'_2) = [0,\ 0] \nabla [0,\ +\infty] = [0,\ +\infty] \end{aligned}$$

This sequence stabilises after the third iteration because $a'_3 = f^\sharp(a'_2) = [0,\ +\infty] = a'_2$; the computed limit $\bigsqcup_{n \geq 0} a'_n = a'_2 = [0,\ +\infty]$ is a sound approximation to the range of values of i.

Note that this example is atypical because the limits of iterations with and without widening are equal. In general, the use of widening may cause overshooting the limit of the abstract iteration; in any case, it still yields a safe upper-bound approximation.

2.3.6 Lattice of convex polyhedra

An interval approximates a single set of integers; to obtain an analysis for multiple components (e.g. ranges of two or more variables) it is possible to use products of intervals, one for each component; this kind of combination is designated an *independent attribute* combination because it does capture any interplay between components (Nielson et al. 1999, pages 249–250), e.g. it does not capture relations between variables, as the following example illustrates.

Example 2.5 Consider the following imperative program:

```
$i := 0;\ j := 1;\ c := \mathbf{true}$
$\mathbf{while}\ c\ \mathbf{do}$
    $i := i + 1$
    $j := j + 2$
    $\mathbf{read}(c)$
```

The abstract interpretation of the ranges of i and j using a product of two interval **Interval** × **Interval** will obtain the ranges $i \in [0, +\infty]$ and $j \in [1, +\infty]$ but no relation between i and j. □

To obtain a *relational* analysis, it is necessary to consider a more expressive abstract domain. One of the most successful approaches for this purpose is to use finite systems of linear inequalities; such systems represent (possibly infinite) convex polyhedral regions in an n-dimensional vector space, or simply, *convex polyhedra*.

Although the use of convex polyhedra in program analysis dates back to one of the earliest papers on abstract interpretation (Cousot and Halbwachs 1978), this domain is still the basis for many state-of-art verification and analysis tools in use today. Moreover, there has recently been a renewed interest in the use of convex polyhedra for program analysis motivated by both theoretical extensions (Bagnara, Hill and Zaffanella 2002, Bagnara, Hill, Ricci and Zaffanella 2003) and the availability of practical implementations, e.g. the *Parma Polyhedra Library* (Bagnara, Ricci, Zaffanella and Hill 2002).

A *closed convex polyhedron*, or simply a *polyhedron*, is the solution-set $P \subseteq \mathbb{R}^n$ of a system of linear inequations (Schrijver 1986)

$$P = \{\, x \in \mathbb{R}^n \ :\ Ax \le b \,\} \tag{2.29}$$

where $A \in \mathbb{Q}^{m \times n}$ is a matrix of coefficients and $b \in \mathbb{Q}^m$ is a vector of constants.[4] The set $\mathbb{CP}_n$ of convex polyhedra of dimension n ordered by $\subseteq$ is a lattice:

[4] Note that we intentionally restrict coefficients to *rational* rather than real numbers to ensure that these are machine representable.

- the intersection $P \cap Q$ of two polyhedra P, Q is a polyhedron (the conjunction of the two systems of constraints);
- the union of two convex polyhedra is not necessarily convex, so the least upper bound of P, Q is not $P \cup Q$; instead it is the *convex hull* $P \uplus Q$, i.e. the smallest polyhedron that contains both P and Q. In general $P \cup Q \subseteq P \uplus Q$ (and the inclusion is strict when $P \cup Q$ is not a convex set).
- the empty set $\emptyset$ and the universe $\mathbb{R}^n$ are, respectively, the bottom and top elements of $\mathbb{CP}_n$.

We remark that $\mathbb{CP}_n$ is *not* a complete lattice: for example, the sphere is not a polyhedron but can be obtained as the limit of an infinite sequence of polyhedra.

The dual description method

We say that (A, b) of equation (2.29) is a *system of constraints* for P. A polyhedron can alternatively be characterised by a *system of generators* (V, R) as the sum of a convex combination of vertices $V = \{v_i \in \mathbb{Q}^n\}$ with a positive combination of rays $R = \{r_j \in \mathbb{Q}^n\}$,

$$P = \left\{ \sum_{i=1}^{|V|} \lambda_i v_i + \sum_{j=1}^{|R|} \mu_j r_j \ : \ \lambda_i \geq 0,\ \mu_j \geq 0,\ \sum_{i=1}^{|V|} \lambda_i = 1 \right\} \tag{2.30}$$

The two descriptions (2.29) and (2.30) are dual of each other in the sense that either one represents the polyhedron and that a single algorithm can switch between representations (Motzkin et al. 1953, Chernikova 1968, Verge 1992).

The dual description method represents polyhedra both by constraints and generators. This is justified because some operations are more efficient on the constraints while others are more efficient on the generators; others still benefit from *both* representations. Another important property is that the duality allows keeping the representations minimal, i.e. free of redundant constraints or generators. Efficient implementations of polyhedra computations are based on the dual description method, taking special care to avoid unnecessary conversions (Wilde 1993, Bagnara et al. 2006).

Operations on polyhedra

We briefly describe the more common computations on polyhedra, mainly to fix notation. For a thorough description we point the reader to (Bagnara et al. 2006). Let P and Q be two polyhedra of n dimensions $x_1, \ldots, x_n$. The following operations are all computationally effective:

Containment test: $P \subseteq Q$ holds if and only if the system of generators of P satisfies the constraints of Q.

Intersection: the system of constraints for $P \cap Q$ is obtained as the union of the constraints for P with those for Q.

Convex hull: the system of generators of $P \uplus Q$ is obtained as the union of the generators of P with those of Q.

Variable elimination: $\mathsf{ELIM}(x_i, P)$ is the polyhedron resultant from eliminating the dimension x_i from P by Fourier elimination (Chandru 1993); the system generators of $\mathsf{ELIM}(x_i, P)$ is obtained by adding two rays $\{x_i, -x_i\}$ to the system of generators of P.

Widening: if $P \subseteq Q$ then $P \nabla Q$ is the *widening* of P and Q. The standard widening for convex polyhedra is due to Halbwachs (1979); more recently, Bagnara, Hill, Ricci and Zaffanella (2003) proposed a more precise widening operator.

Widening operators for convex polyhedra

Since the lattice of convex polyhedra is incomplete, we need to employ a widening operator to guarantee termination of fixed point approximation (see Section 2.3.3).

The first widening operator for convex polyhedra was proposed by Cousot and Halbwachs (1978) for synthesising loop invariants of imperative programs and later formalised by Halbwachs (1979) in his PhD thesis. Informally, $P \nabla Q$ is the set of constraints of P that are still satisfied by Q. This is an upper bound operator because $P \nabla Q$ is defined by a subset of the constraints of both polyhedra. It is a widening because the system of constraints of P is finite, therefore it is not possible to keep removing constraints indefinitely.[5]

Example 2.6 Consider again the imperative program of Example 2.5:

```
$i := 0;\ j := 1;\ c :=$ true
while $c$ do
    $i := i + 1$
    $j := j + 2$
    read($c$)
```

[5] The formal definition is slightly more elaborate to make the widening well-defined for equivalent but syntactically-distinct constraint systems; see (Halbwachs 1979, Bagnara, Hill, Ricci and Zaffanella 2003) for the details.

We will perform abstract interpretation using $\mathbb{CP}_2$ to determine loop invariants as linear inequalities in two dimensions. For readability, we represent elements of $\mathbb{CP}_2$ by systems of linear inequations using the same variable names i, j as in the program.

The abstract transition function should approximate the effect of one loop iteration: given a constraint system X describing (some) reachable values of i, j, then $f^\sharp(X)$ should describe the reachable values after one more iteration. It is possible to reach the loop body from either the begining of the program or by remaining in the loop after the test condition; hence, we can define $f^\sharp(X)$ using the least upper bound of the two computational paths:

$$\begin{aligned} f^\sharp &: \mathbb{CP}_2 \to \mathbb{CP}_2 \\ f^\sharp(X) &= \{i = 0,\ j = 1\} \uplus \{(i+1,\ j+2) : (i,j) \in X\} \end{aligned}$$

The constraints $\{i = 0,\ j = 1\}$ in the definition above encode the exact values of the variables at the begining of the program; the constraints $\{(i+1,\ j+2)\} : (i,j) \in X\}$ encode the effects of the assignments done in the loop body.[6]

We can now express an approximation of reachable values as the limit of sucessive iterates $a_0 = \emptyset$ and $a_{n+1} = f^\sharp(a_n)$. The first iterates are as follows:

$$\begin{aligned} a_0 &= \emptyset \\ a_1 = f^\sharp(a_0) &= \{i = 0,\ j = 1\} \uplus \emptyset && = \{i = 0,\ j = 1\} \\ a_2 = f^\sharp(a_1) &= \{i = 0,\ j = 1\} \uplus \{i = 1,\ j = 3\} && = \{0 \leq i \leq 1,\ j = 1 + 2i\} \\ a_3 = f^\sharp(a_2) &= \{i = 0,\ j = 1\} \uplus \{1 \leq i \leq 2,\ j = 1 + 2i\} && = \{0 \leq i \leq 2,\ j = 1 + 2i\} \\ a_4 = f^\sharp(a_3) &= \{i = 0,\ j = 1\} \uplus \{1 \leq i \leq 3,\ j = 1 + 2i\} && = \{0 \leq i \leq 3,\ j = 1 + 2i\} \\ &\vdots \end{aligned}$$

It is immediate that the sucessive iterates are strictly increasing and will not stabilise in a finite number of steps.

As in the case of intervals, we can force finite convergence by employing a widening operator ∇. Here we employ Halbwachs' widening after the second iterate:

$$\begin{aligned} a'_3 = a_2 \nabla f^\sharp(a_2) &= \{0 \leq i \leq 1,\ j = 1 + 2i\} \nabla \{0 \leq i \leq 2,\ j = 1 + 2i\} \\ &= \{0 \leq i,\ j = 1 + 2i\} \end{aligned}$$

The widening operator discarded a single unstable constraint $i \leq 1$. We can now easily verify that the a'_3 is a sound loop invariant because $f^\sharp(a'_3) = \{i = 0,\ j = 1\} \uplus \{1 \leq i,\ j = 1 + 2i\} = \{0 \leq i,\ j = 1 + 2i\} = a'_3$, i.e. the iteration has stabilised.

[6] Note that this must be a linear constraint system when X is, because it is defined by a linear translation of X.

Although this is not always the case, note that we have obtained the exact limit $\biguplus_{n\geq 0} a_n$ of the iterates above.

In general, the iteration might not stabilise after a single application of the widening; the process must be repeated until a post fixed point is reached (this must happen in a finite number of steps by the properties of the widening).

We obtain a system of two linear constraints $0 \leq i$, $j = 1 + 2i$. In particular, the linear relation $j = 1 + 2i$ between the two variables is much could not obtained by the interval analysis of Example 2.5. Note also that we do not loose precision: the lower bound for j is a consequence of the two infered linear constraints. □

Halbwachs' widening has been used in most analysis tools employing convex polyhedra and has received the designation of the *standard widening* for convex polyhedra. However, this widening is sometimes too coarse, loosing many constraints before stabilising.

Several techniques have been proposed in the abstract interpretation literature to improve the precision of iteration with widening. One approach is simply to not use the widening of the first k iterations, where k is some fixed small constant. This is in fact what we did in Example 2.6: the widening was not applied until the third iteration. Delaying the application of widening allows accumulating more information during the first iterations; convergence is still ensured from the k-th iteration onwards.

A more precise variant is the "widening with tokens" of Bagnara, Hill, Ricci and Zaffanella (2003). The iteration starts with a fixed number of *tokens*; one token is consumed each time the widening would cause a loss of precision and the exact least upper bound is used instead; the standard widening is used when there are no tokens left. The advantage of this technique is that the number of initial tokens specify the delaying of *actual* rather than *potential* losses of precision.

A final approach is to choose a more precise widening operator; this is highly dependent on the lattice of abstract properties and little can be said about how to proceed in general. Halbwachs' widening was the sole proposal for abstract interpretation using convex polyhedra from its inset in the late 1970s for over 20 years. More recently, Bagnara, Hill, Ricci and Zaffanella (2003) proposed a new widening operator for convex polyhedra which they proved to be no less precise than Halbwachs' widening in the worst-case and more precise in some cases.

Chapter 3

Static analysis for time and space costs

The problem of determining bounds for time or space usage of functional programs has been extensively addressed in the literature using various approaches. In this chapter we present a review of the most relevant work.

In order to highlight connections and present a clearer view of the field, we group previous work into separate sections: automatic complexity analysis (Section 3.1), type and effect systems for time (Section 3.2), sized types (Section 3.3), dependent types (Section 3.4), amortised analysis (Section 3.5). Finally, we review indirectly related work in Section 3.6.

3.1 Automatic complexity analysis

Early works in automatic cost analysis follow the methodology for hand analysis of algorithms, e.g. the seminal textbook by Knuth (1973): first derive some recurrence equations expressing the program cost (e.g. number of arithmetic or other primitive operations) in terms of an input metric (e.g. data size) and then solve the recurrences (perhaps using approximation) to obtain a closed equation.

The earliest work following this methodology is Wegbreit's METRIC system (Wegbreit 1975). METRIC derived complexity equations for list functions written in a first-order subset of LISP with recursive procedures, but no side-effects or imperative features. The system obtained metrics such as time, length or size as a 4-tuple $\langle min, max, avg, var \rangle$ of lower bound, upper bound, average and standard deviation; the first two are best and worst-case bounds; the last two measures are derived under the assumption of statistical independence of dynamic tests. The performance measures are expressed symbolically as functions of input size or length and the costs of primitive operations.

METRIC first transformed a recursive function into a step-counting version, i.e. a function with the same domain and whose value is the cost metric (here "cost" can be length, size or time, e.g. number of reduction steps). As an example, consider a function appending two lists:[1]

$$\begin{aligned}\mathit{APPEND}(X,Y) = &\text{if NULL}(X) \text{ then } Y\\ &\text{else CONS}(\text{CAR}(X),\ \mathit{APPEND}(\text{CDR}(X),Y))\end{aligned}$$

The step counting function for *APPEND* under the time metric produced the symbolic cost function

$$\begin{aligned}\mathit{time}(\mathit{APPEND}(X,Y)) = &\text{if NULL}(X) \text{ then } \mathit{null} + 2\mathit{vref}\\ &\text{else } \mathit{null} + 4\mathit{vref} + \mathit{cdr} + \mathit{car} + \mathit{cons} +\\ &\quad \mathit{time}(\mathit{APPEND}(\text{CDR}(X),\ Y))\end{aligned}$$

where *null*, *car*, *cdr* and *cons* are symbolic constants for the costs for the primitive list operations and *vref* is the cost for accessing a variable.

The system now projects the step-counting function into a recurrence equation over the integers, using either a measure of the *size* (total number of nodes) or *length* (number of nodes along the *cdr*-direction) of arguments. For this example, the recursion involves only the *cdr* of the first argument, so METRIC chooses the length measure and expresses

$$\begin{aligned}\mathit{time}(\mathit{APPEND}(X,Y)) &= T(\mathit{length}(X))\\ T(0) &= \mathit{null} + 2\mathit{vref}\\ T(n+1) &= \mathit{null} + 4\mathit{vref} + \mathit{cdr} + \mathit{car} + \mathit{cons} + T(n)\end{aligned}$$

The system then attempts to solve the recurrence equation to obtain a closed-form expression. The solution obtained for this example is:

$$\begin{aligned}\mathit{time}(\mathit{APPEND}(X,Y)) = &\mathit{null} + 2\mathit{vref} +\\ &(\mathit{null} + 4\mathit{vref} + \mathit{cdr} + \mathit{car} + \mathit{cons}) \times \mathit{length}(X)\end{aligned}$$

Note that the closed cost equatiom expresses not just the asymptotic complexity (append is linear on the length of the first argument) but also the constants terms involved as a combination of the parameters *null*, *vref*, etc.

[1] In this example we use LISP constructors names: *CONS* is the pair constructor; *CAR* and *CDR* project the first and second element of a pair; *NIL* is a constant; and *NULL* tests equality to the *NIL* constant.

We remark also that this example is uncharacteristically simple: in general to analyse a function under one measure, METRIC might have to perform sub-analysis under other measures (e.g. length or size of sub-expressions). Solving recurrence equations with more alternatives also requires more sophisticated techniques, e.g. the use of generating functions.

METRIC was able to obtain closed cost equations for simple LISP programs, e.g. list reverse, flattening, membership test and union. The analysis could also be used to predict heap allocation by counting the number of *cons* instructions executed. However, it could not be used to predict non-cumulative metrics such as stack depth because of the assumption that costs are additive:

$$\mathit{time}(F(G(X))) = \mathit{time}(G(X)) + \mathit{time}(F(Y))\,, \quad \text{where } Y = G(X)$$

While the above holds for monotonically increasing resources such as time[2] or (total) heap allocation, it does not model the reuse of stack space. The combination of stack costs should be (ignoring the overheads for function applications, and once again assuming call-by-value reduction):

$$\mathit{stack}(F(G(X)) = \max(\mathit{stack}(G(X)),\, \mathit{stack}(F(Y))\,, \quad \text{where } Y = G(X)$$

The stack cost for the composition is the *maximum* of the stack used for the sub-expression and outer call. While in theory it suffices to synthesise recurrences with maximums instead of sums, in practice such recurrences are much harder to solve automatically because maximum is not an analytic function.

METRIC was also limited to list processing: the complexity bounds are derived with respect to either the length or the size of S-expressions; the system chooses one of the two measures using an heuristic based on the use of arguments in recursive calls. There is no way to use specialised measures as would be desirable, e.g. for user-defined data types.

Le Métayer's ACE system also performed complexity analysis by deriving a recursive step-counting function from each recursive function (Le Métayer 1988). However, a first departure from the work by Wegbreit is that the costs measure considered is *worst-case* only and *asymptotic*; this means that individual primitive operations are not accounted, only the number of recursive calls.

Second, unlike Wegbreit's approach of projecting the cost function on the integers and solving recurrence equations, ACE obtained closed-form solutions by a series of meaning-preserving program transformations within a functional calculus (a subset

[2] But is accurate only under *call-by-value* reduction strategy; see (Wadler 1988) for a formalisation of the corresponding assumption for *call-by-need* strategy, i.e. lazy evaluation.

of the FP language). The transformations are based on an algebra of applicative program transformations together with McCarthy's recursion induction principle: two functions satisfying the same recurrence equation are equal.[3]

For example, the factorial function can be expressed in FP as follows:

$$\text{fact} = \text{eq0} \rightarrow \text{“1”}; *o[\text{id}, \text{fact}\, o\, \text{sub1}]$$

The definition is in point-free style: $f \rightarrow g; h$ is a conditional with test f and true and false branches g and h; eq0 tests the argument for zero; id is the identity function; o is function composition; "k" is the constant k-valued function; sub1 is the integer predecessor function; $[f_1, \ldots, f_k]$ builds a sequence by applying f_i to an argument; $*$ and $+$ are arithmetic operators which operate on sequences of two values.

The step-counting function *Cfact* mimics the recursion structure of *fact* but adds one unit of cost for each call and zero for constants and primitives:

$$\begin{aligned}\text{Cfact} =&\text{eq0} \rightarrow +o[\text{“0”}, \text{”0”}];\\ &\qquad + o[\text{“0”}, +o[\text{“0”} \; + o[\text{“0”}, +o[\text{plus1}\, o\, \text{Cfact}\, o\, \text{sub1}, \; \text{“0”}]]]]\end{aligned}$$

To obtain a closed-form solution for the recursive *Cfact*, ACE employs a number of program transformations expressed as rewrite rules. For example, using the rules $+o[\text{“0”}, f] = +o[f\, \text{”0”}] = f$ (i.e. zero is the neutral element for sum) the definition of *Cfact* can be simplified to:

$$\text{Cfact} = \text{eq0} \rightarrow \text{“0”}; \text{plus1}\, o\, \text{Cfact}\, o\, \text{sub1}$$

Using the recursion induction principle, the above definition is matched against a library to find

$$\text{Cfact} = \text{id}$$

and conclude that the complexity of factorial is linear in the argument value.

Le Métayer reports success in analysing numerical programs, sorting algorithms and a parser. However, no indication is given as to the quality of results. Moreover, the quality of results appear to be very sensitive to the set of rewrite rules provided: the implementation is said to use over 1,000 rules of various kinds. The extent to which these match specific programs or general programming patterns is not discussed.

The system deals with time in a very abstract manner (since only function calls are accounted) and not heap space or stack depth. The same limitations regarding cumulative cost measure that apply to Wegbreit's METRIC hold here. Asymptotic

[3] On the domain of the least solution of the recurrence.

results would, in any case, be insufficient for bounding time or space in high integrity systems because the lower order terms might dominate the actual worst-case.

Another severe limitation for domains where high integrity is required is that a single incorrect rule in the database compromises the soundness of the results. This is even more problematic if the user is allowed to extend the system with new axioms and rules for specific programs.

Le Métayer points out that the complexity functions resulting from ACE can sometimes be several lines long; such a result would be unintelligible for a human reader and possibly unusable by a compiler. Finally, the ACE system is very tightly coupled with a particular language: algorithms that are not naturally expressed in FP are very difficult to analyse in ACE.

Rosendahl (1989) describes a semantic-based method for deriving the step-counting version of recursive first-order functions. The main contribution is the use of abstract interpretation to define a time-bound function whose inputs are partial representations of the original program inputs and whose output is an upper-bound on the original program time. No attempt is made to obtain closed cost expressions.

Liu and Gomez (1998) have also presented an automatic time analysis for a first-order subset of Lisp based on obtaining a time-bound function on partial representation of inputs. Instead of trying to obtain closed-form solutions, they use the time-bound function to compute upper bounds. For example, to obtain a bound for sorting a list of n values, they symbolically execute the time-bound function with a list of n "unknown" LISP atoms. Unnikrishnan, Stoller and Liu (2000) have applied this technique to obtain bounds on stack and heap costs.

The limitation of this approach is that it does not yield a closed cost expression: if the original function is recursive, so is the time-bound function. Moreover, symbolic execution of the time-bound program with a partial input will not terminate if the recursion involves an unknown term. To ensure termination the input must be of fixed size (e.g. a list of 10 unknown values); the time-bound function then returns the cost for that specific size. Thus, this approach is closer to profiling than static analysis.

Furthermore, the time-bound function can be exponentially more expensive to compute than the original program, since it has to execute both branches of conditionals that depend on unknowns. This leads to performance problems with even moderate size inputs: Unnikrishnan et al. report a running time exceeding 2 hours to obtain upper bounds for merge sort of 30 elements and were unable to obtain bounds for 1,000 elements.

All works in complexity analysis described so far (Wegbreit 1975, Le Métayer 1988, Rosendahl 1989) assumed a cumulative measure of time cost. This is a very coarse overestimate under lazy evaluation, since the arguments may be only partially evaluated. The problem of *compositional* time analysis for lazy evaluation is that the time cost for an expression depends on the context, i.e. the amount of the input that is "needed" by each function.

Wadler (1988) proposed a formalism for time analysis for first-order lazy evaluation using *projection transformers* to capture the "neediness" of each function. This work presented a formalism for expressing the step-counting equations but not an algorithm for approximating them.

Bjerner and Holmström (1989) also proposed a time analysis for first-order lazy functional programs. This approach is based on an abstract representation of demand in result values; they then perform a backwards demand analysis to find out how much of the input is required to produce the required output. One limitation is that the representation of a demand requires knowing in advance much information about the output value.

In his PhD thesis, Sands (1990) developed several calculi for time analysis of functional programs, including the treatment of higher-order functions and lazy evaluation. His approach was to employ program transformations to derive costs functions from the original program. To deal with higher-order function, Sands proposed the use of *cost-closures*. Cost-closures are structures that pair each function with its corresponding cost function. He also refined the approach of Wadler (1988) by employing strictness information to obtain *sufficient-time* and *necessary-time* cost equations for lazy functions.

All the above formalisms (Wadler 1988, Bjerner and Holmström 1989, Sands 1990) are intended to aid a human in reasoning about program costs, but are not directly automatable for use in a compiler or verification tool. Moreover, the model of time cost considered is asymptotic (e.g. number of non-primitive function calls) and consequently not directly related to the implementation time or space costs.

3.2 Type and effect systems for time

Dornic, Jouvelot and Gifford (1992) presented a polymorphic "time system" for deriving time costs for higher-order call-by-value functional language. This system is an instance of a type and effect analysis where an underlying type system is extended with "effects" that approximate some intentional property of evaluation (Jouvelot and Gifford 1991, Talpin and Jouvelot 1992, 1994). In the time system, effects approximate the number of computation steps needed to reduce an expression to normal form.

The starting point is the simply typed lambda-calculus with a strict (i.e. call-by-value) semantics.[4] A type judgement $\Gamma \vdash e : \tau$, where Γ is a typing context, e is a term and τ is a type, is augmented with an *effect* n:

$$\Gamma \vdash e : \tau \ \$ \ n \tag{3.1}$$

The effect n is either a natural number representing an upper-bound on the number of reduction steps for e or a distinguished element `long` representing a potentially unbounded reduction. Thus, the augmented judgement (3.1) reads: *under assumptions* Γ*,* e *has type* τ *and cost* n*.*

As in other type and effect systems, functional types in Dornic's time system are annotated with a "latent effect" (designated *latent cost* in this system) that expresses the cost of function evaluation. The latent cost mechanism allows capturing the cost of application of higher-order functions quite naturally, as can be seen in the type rule for application:

$$\frac{\Gamma \vdash e_0 : \tau' \xrightarrow{l} \tau \ \$ \ m \qquad \Gamma \vdash e_1 : \tau' \ \$ \ n}{\Gamma \vdash (e_0\ e_1) : \tau \ \$ \ m + n + l + 1} \tag{3.2}$$

The type rule expresses the cost of an application $(e_0\ e_1)$ as a sum of: the cost m of reducing e_0 to some lambda-abstraction $\lambda x.\ e'$; the cost n of reducing the argument e_1 to a normal form v; the latent cost l of reducing the *β-reduct* $[x \mapsto v]\, e'$; and one extra unit to account for the application itself.

The dual rule for lambda-abstraction transposes the actual cost of a function body into a latent one:

$$\frac{\Gamma,\, x : \tau' \vdash e : \tau \ \$ \ m}{\Gamma \vdash (\lambda x.\, e) : \tau' \xrightarrow{m} \tau \ \$ \ 1} \tag{3.3}$$

Rules (3.2) and (3.3) reflect the chosen cost model: each application, lambda-abstraction and variable access cost one unit. Different choices could easily be accommodated by choosing different constants in the type rules.

Using these rules we can derive a "timed" type for the higher-order term *twice* $\equiv$

[4] To simplify the presentation, we use the lambda-calculus rather than the *CT* language of (Dornic et al. 1992).

$\lambda f.\, \lambda x.\, f\ (f\ x)$ as follows:

$$
\begin{array}{lr}
f : \tau \xrightarrow{l} \tau,\, x : \tau \vdash x : \tau \ \$\ 1 & \{\text{hypothesis}\} \\
f : \tau \xrightarrow{l} \tau,\, x : \tau \vdash f : \tau \xrightarrow{l} \tau \ \$\ 1 & \{\text{hypothesis}\} \\
f : \tau \xrightarrow{l} \tau,\, x : \tau \vdash (f\ x) : \tau \ \$\ 3 + l & \{\text{application, arithmetic}\} \\
f : \tau \xrightarrow{l} \tau,\, x : \tau \vdash (f\ (f\ x)) : \tau \ \$\ 5 + 2l & \{\text{application, arithmetic}\} \\
f : \tau \xrightarrow{l} \tau \vdash \lambda x.\, (f\ (f\ x)) : \tau \xrightarrow{5+2l} \tau \ \$\ 1 & \{\text{abstraction}\} \\
\vdash \lambda f.\, \lambda x.\, (f\ (f\ x)) : (\tau \xrightarrow{l} \tau) \xrightarrow{1} \tau \xrightarrow{5+2l} \tau \ \$\ 1 & \{\text{abstraction}\}
\end{array}
$$

We can interpret the inferred type as follows: *twice* takes a function of type $\tau \to \tau$ and cost l and yields a function of the same type and cost $5 + 2l$; this latent cost expresses the duplicate reduction of the argument function; the added constant corresponds to the cost of three variable references and the two applications.

The full power of the time system is only obtained when the language is extended with polymorphism. Dornic et al. introduce polymorphism via a "polymorphic lambda"; we will do so using let-bound polymorphism. In an expression

$$\mathsf{let}\ \mathit{twice} = \lambda f.\, \lambda x.\, f\ (f\ x)\ \mathsf{in}\ \ldots$$

the identifier *twice* can be given a type quantified over type *and* cost variables:

$$\forall a.\, \forall l.\, (a \xrightarrow{l} a) \xrightarrow{1} a \xrightarrow{5+2l} a$$

Note that the latent cost l is now a quantified variable; this means that the analysis is parametric, i.e. distinct uses of *twice* in the program can be typed with different costs. Moreover, it does not require the whole program: the quantified type of *twice* captures all information needed for future uses, so it is possible to perform separate analysis of libraries and modules.

However, the time system has some important limitations: first, recursive functions are always assigned the unbounded cost `long`. This is because the cost of a recursive function depends on the *sizes* of arguments which are not captured in the time system. The absence of size information also severely limits the precision of the analysis of higher-order functions, since the costs cannot depend on the sizes of arguments.

Second, the type system does not allow *subeffecting*, i.e. subsuming a cost by a larger one; this is needed, e.g. to be able to type a conditional with different costs in each branch; the extension (adding a maximum function to the cost algebra) is proposed as further work.

Finally, Dornic et al. considered only *checking* timed types, i.e. all time information must be prescribed as type annotations; the problem of *reconstructing* timed types was address in the subsequent work by Reistad and Gifford (1994).

Reistad and Gifford extended the time system of Dornic et al. with annotations representing sizes of naturals, lists and vector types and with an algorithm to reconstruct sizes and times based on algebraic reconstruction of effects (Jouvelot and Gifford 1991). This system has been applied to aid dynamically scheduling in a parallelising implementation of the μFX language.

The size annotations represent upper bounds on the dynamic sizes of values. For example, the values of a type $\mathsf{Nat}\ n$ are the naturals less-than or equal to n. More interestingly, annotations in function types describe size changes, e.g. the type for the successor function is

$$\mathit{succ} : \forall n.\, \mathsf{Nat}\ n \xrightarrow{1} \mathsf{Nat}\ (n+1)$$

assuming a cost of one unit for the operation. The algebra for sizes and costs includes a value `long` to represent a potentially unbounded sizes, and operations of addition, maximum and multiplication. This *sized* timed type system allows subsuming sizes: for example, the typing rule for natural constants is

$$\frac{\mathit{nat} \leq n}{\Gamma \vdash \mathit{nat} : \mathsf{Nat}\ n\ \$\ C_{num}}$$

where C_{num} is cost associated with naturals. Thus, the natural 1 admits any type $\mathsf{Nat}\ n$ for $1 \leq n$ (including `long`). Without this flexibility a conditional expression like "$\mathsf{if}\ \ldots\ \mathsf{then}\ 1\ \mathsf{else}\ 2$" would not admit a type because $\mathsf{Nat}\ 1 \neq \mathsf{Nat}\ 2$. The ordering relation $\leq$ on sizes induces a structural subtyping relation $\leqslant$ on the annotated types (Mitchell 1984).

Adding size information to types can also allow specifying more precise costs. For example, the higher-order *map* function can be assigned the following type:

$$\mathit{map} : \forall\{a,\, b,\, c,\, l\}.\, (a \xrightarrow{c} b) \times \mathsf{List}\ a\ l \xrightarrow{k_0 + l \times (k_1 + c)} \mathsf{List}\ b\ l$$

Note that the type for *map* expresses not just that the result list has the same length l as the input, but also the cost of *map* as a function of the argument cost c and list length l.[5] Such dependency was not possible in the time system of Dornic et al. because of the absence of size information.

The main limitation of this work is the absence of a treatment of recursion. As in the time system of Dornic et al. recursive functions can only be typed with a `long`

[5] Constants k_0 and k_1 must be chosen to reflect the overheads associated with a particular implementation.

cost. To mitigate this, Reistad and Gifford use a fixed set of higher-order functions (such as *map* above) to express primitive recursion schemes.

A minor limitation of the size semantics is that sizes of non-increasing functions must be overestimated. For example, a subtraction operator on the naturals must be given the type

$$sub : \mathsf{Nat}\ n \times \mathsf{Nat}\ m \xrightarrow{C_{sub}} \mathsf{Nat}\ n$$

because we only know that the second argument is at most m (in particular, it could be zero). One solution to this problem would be to extend the size semantics to include lower bounds as well as upper bounds, i.e. intervals.

Another limitation of the system of Reistad and Gifford is an overestimation of sizes caused by insufficient polymorphism in higher-order functions. Consider the following sized timed types[6] for the higher-order function *twice* and the natural successor:

$$\begin{aligned} &twice : \forall a.\, \forall b.\, \forall l.\, \langle (a \xrightarrow{l} b) \xrightarrow{1} a \xrightarrow{5+2l} b,\, b \leqslant a \rangle \\ &succ : \forall n.\, \mathsf{Nat}\ n \xrightarrow{1} \mathsf{Nat}\ (n+1) \end{aligned}$$

To type the application (*twice succ*) we must solve the subtyping constraints:

$$\begin{aligned} \mathsf{Nat}\ n \xrightarrow{1} \mathsf{Nat}\ (n+1) &\leqslant a \xrightarrow{l} b \\ b &\leqslant a \end{aligned}$$

Decomposing the subtyping constraints we get (note the contravariance on the left-side of the arrow):

$$\begin{aligned} a &\leqslant \mathsf{Nat}\ n \\ \mathsf{Nat}\ (n+1) &\leqslant b \\ b &\leqslant a \\ 1 &\leq l \end{aligned}$$

Since the subtyping is shape conformant, we can now substitute $a \equiv \mathsf{Nat}\ i$ and $b \equiv \mathsf{Nat}\ j$ for some sizes variables i, j and obtain the size inequations $i \leq n \wedge n + 1 \leq j \wedge 1 \leq l \wedge j \leq i$. It is straightforward to check that the only solution is $n = i = j = \texttt{long}$ and $7 \leq l$; thus, the application must be typed as:

$$twice\ succ : \mathsf{Nat}\ \texttt{long} \xrightarrow{7} \mathsf{Nat}\ \texttt{long}$$

[6] We present the type scheme for *twice* in a more general form than that of Reistad and Gifford by allowing subtyping constraints in quantified types (Mitchell 1984, Fuh and Mishra 1988). This is done in order to stress that the problem is due to insufficient polymorphism rather than insufficient subtyping.

Although the latent cost for the result function is accurate, no size information is known.

This problem was designated *size aliasing* in (Portillo et al. 2003) and is caused by the use of a monomorphic type at two distinct sizes. One solution is to extend the type system with discrete polymorphism, i.e. intersection types (Simões et al. 2007). In such a system *twice* admits types of the form

$$\mathit{twice} : (\tau \xrightarrow{l} \tau' \wedge \tau' \xrightarrow{l'} \tau'') \xrightarrow{1} \tau \xrightarrow{5+l+l'} \tau''$$

By instantiation, *succ* admits the types

$$\mathit{succ} : \mathsf{Nat}\ n \xrightarrow{1} \mathsf{Nat}\ (n+1)$$
$$\mathit{succ} : \mathsf{Nat}\ (n+1) \xrightarrow{1} \mathsf{Nat}\ (n+2)$$

and therefore

$$\mathit{succ} : (\mathsf{Nat}\ n \xrightarrow{1} \mathsf{Nat}\ (n+1)) \wedge (\mathsf{Nat}\ (n+1) \xrightarrow{1} \mathsf{Nat}\ (n+2))$$

Finally, *twice succ* can be typed as

$$\mathit{twice\ succ} : \mathsf{Nat}\ n \xrightarrow{7} \mathsf{Nat}\ (n+1)$$

which accurately expresses both the size and time of the application.

Loidl (1998) proposed a type analysis with size and time information for aiding the scheduling of tasks in a parallel implementation of functional languages by providing static granularity information. He proposes extending the size and time analysis of Reistad and Gifford (1994) to recursive functions by synthesising recurrence equations; these can then be solved to obtain closed-form cost equations either manually or with the aid of a computer algebra system. To make the analysis automatic, Loidl proposes building a library of known recurrence forms, as was done by Rosendahl (1989).

This size and cost analysis was further developed by Vasconcelos and Hammond (2004), which also presented results from a prototype implementation. The difficulties that apply to other approaches based on synthesising recurrences (Wegbreit 1975, Le Métayer 1988) hold here: obtaining approximate solutions to recurrence equations automatically is difficult; the use of a library of recurrences mitigates this problem to some extent, but for practical use this must contain a large number of distinct but similar recurrences; furthermore, a single incorrect assumption in this library invalidates the soundness of the analysis, which is particularly relevant for our intended application domain of embedded and real-time systems.[7]

[7] It is worth remarking that soundness of cost approximations is not critical for the granularity analysis of Loidl, since erroneous cost information could reduce performance but not cause failure.

A further limitation is the loss of precision with irregular divide-and-conquer recursions, e.g. as in the quicksort algorithm. This is shared by other type analysis where the sizes of data components are independently approximated and will discussed in more detail in the next section.

3.3 Sized types

Some researchers have presented type based analysis for size information alone; this can be useful for proving termination, enabling optimisations in compilers (e.g. eliminating array bounds checks) or for enabling program transformations (e.g. partial evaluation).

Hughes, Pareto and Sabry (1996) presented a type system extended with size information for proving liveness properties of reactive systems, namely termination and productivity.

The term language considered is purely-functional, non-strict and higher-order with let-bound polymorphism, general recursion and algebraic data types. The sized type system of Hughes et al. distinguishes *data* values (e.g. naturals or finite lists) from *codata* values (e.g. streams): the size of a data value is an upper bound on the number of constructors, while for a codata value it is a *lower* bound. For example, given the declarations for naturals, finite lists and infinite lists (streams),

$$\begin{aligned} \mathsf{idata\ Nat} &= \mathsf{Zero} \ \mid \ \mathsf{Succ\ Nat} \\ \mathsf{idata\ List}\ a &= \mathsf{Nil} \ \mid \ \mathsf{Cons}\ a\ (\mathsf{List}\ a) \\ \mathsf{codata\ Stream}\ a &= \mathsf{Mk}\ a\ (\mathsf{Stream}\ a) \end{aligned}$$

the corresponding sized types for constructors are:

$$\begin{aligned} \mathsf{Zero} &: \mathsf{Nat}_1 \\ \mathsf{Succ} &: \forall i.\, \mathsf{Nat}_i \to \mathsf{Nat}_{i+1} \\ \mathsf{Nil} &: \forall a.\, \mathsf{List}_1\ a \\ \mathsf{Cons} &: \forall i.\, \forall a.\, a \to \mathsf{List}_i\ a \to \mathsf{List}_{i+1}\ a \\ \mathsf{Mk} &: \forall i.\, \forall a.\, a \to \mathsf{Stream}^i\ a \to \mathsf{Stream}^{i+1}\ a \end{aligned}$$

The data types are annotated with a subscript or superscript size annotation (for data or codata, respectively). Size annotations are restricted to arithmetic expressions using constants (natural numbers), variables and addition, but *not* multiplication; this subset of arithmetic can be checked computationally using a Presburger arithmetic solver such as the Omega Calculator (Pugh 1992).[8]

[8] *Presburger arithmetic* is the first-order logic theory of the natural numbers with addition;

The types for Succ, Cons and Mk express size relations: the result has one more constructor than the argument. Note that Zero and Nil have size *one* (not zero) because the size is the number of *constructors* of the value.

Sized data types can be seen as infinite families of approximations indexed by the number of constructors, e.g. $\mathsf{Nat}_0 \subseteq \mathsf{Nat}_1 \subseteq \mathsf{Nat}_2 \subseteq \ldots$. A special annotation ω is used to denote the "limit" of these approximations, e.g. Nat_ω is the type of all naturals.[9] As in the system of Reistad and Gifford, the size ordering induces a structural subtyping relation on sized types. Subtyping is used, for example, to assign a sized type to a conditional with expressions of different sizes in the two branches.

The novelty of the type system of Hughes et al. is a typing rule for recursion that embodies a principle of induction on sizes and that guarantees termination of recursive functions (and dually, productivity of corecursive ones). Omitting type variable generalisations for simplicity, the rule is:

$$\frac{\begin{array}{c} \mathit{all}(\tau[0]) \\ \Gamma \vdash \lambda x.\, M : \forall i.\, \tau[i] \to \tau[i+1] \\ \Gamma \cup \{x : \forall i.\, \tau[i]\} \vdash N : \tau' \end{array}}{\Gamma \vdash \mathsf{letrec}\ x = M\ \mathsf{in}\ N : \tau'} \quad i \notin FV(\Gamma) \tag{3.4}$$

The first two hypotheses express the induction on a size variable i that occurs in a type $\tau[i]$ (we use square brackets for a context):

1. $\tau[0]$ must be a *universal type*, i.e. one that includes the totally undefined value $\bot$;[10]

2. progress must be made at each recursive call, i.e. we must be able to derive $\tau[i+1]$ assuming $\tau[i]$.

To see how rule (3.4) rejects non-terminating functions, consider the following (erroneous) list length function:

$$\mathit{wronglen}\ xs = \mathsf{case}\ xs\ \mathsf{of}\ \mathsf{Nil} \to \mathsf{Zero}\ \mid\ \mathsf{Cons}\ x\ xs' \ \to\ \mathsf{Succ}\ (\mathit{wronglen}\ xs)$$

because it omits multiplication, Presburger arithmetic is less expressive than *Peano arithmetic*. However, the Presburger fragment is decidable (Cooper 1972) while (by Gödel's incompleteness theorem) Peano arithmetic is undecidable.

[9] Note that ω does not represent the absence of size information but rather a special "limit" size. In particular, instantiating a quantified variable with ω is *not* always sound in Hughes and Pareto's system.

[10] The semantics for sized types of Hughes et al. is based on upwards-closed sets rather than the standard semantics based on ideals (MacQueen and Sethi 1982), so that sized types can exclude $\bot$.

This function diverges for non-empty lists because the recursive call is on xs rather than xs'. Type checking against the type

$$\mathit{wronglen} : \forall i.\, \forall a.\, \mathsf{List}_i\ a \to \mathsf{Nat}_i$$

gives rise to the proof obligation

$$\{\, \mathit{wronglen} : \mathsf{List}_i\ a \to \mathsf{Nat}_i,\ xs : \mathsf{List}_{i+1}\ a,\ \ldots \} \vdash \mathsf{Succ}\ (\mathit{wronglen}\ xs) : \mathsf{List}_{i+1}\ a$$

But typing the application ($\mathit{wronglen}\ xs$) requires solving the subtyping constraint $\mathsf{List}_{i+1}\ a \leqslant \mathsf{List}_i\ a$ which is impossible because $i + 1 \nleq i$; the erroneous function is therefore rejected.

The correctness of rule (3.4) was proved using a non-standard type semantics that allows types to exclude $\perp$ (i.e. non-termination/non-productivity). A sketch of the proof was presented in (Hughes et al. 1996); the complete proof together with type checking algorithm was presented by Pareto (1998). The algorithm requires let-bound identifiers (in particular, recursive functions) to be annotated with sizes but infers those of intermediate expressions. The type checker rejects programs whose termination/productivity is not ensured by the sized type annotations provided by the user. By the undecidability of the halting problem, such a decision procedure must reject some terminating/productive programs as well.

The sized type system allows primitive recursive definitions over naturals and lists (e.g. *append*, *map* and *filter*). Functions with an accumulating parameter (e.g. *reverse*) can also be accepted by extending the type rule for recursion with size polymorphism (but not type polymorphism, thus retaining decidability of type checking). More complex recursions can sometimes be re-written so that the type checker will accept them (e.g. the system rejects the usual first-order definition of the Ackermann function, but accepts the higher-order primitive recursive one, because in that version termination is explicit in the structure of the definition).

However, the system has limitations with irregular recursion patterns, e.g. divide-and-conquer algorithms where the data size does not reduce uniformly in recursive calls. Consider the quicksort algorithm for lists, using an auxiliary function *split_by* that breaks up a list into two sub-lists of the smaller and greater elements with respect to a pivot[11]:

$$
\begin{aligned}
&\mathit{qsort} : \mathsf{List}\ t \to \mathsf{List}\ t\\
&\mathit{qsort}\ [\,] = [\,]\\
&\mathit{qsort}\ (x : xs) = \mathsf{case}\ \mathit{split\_by}\ x\ xs\ \mathsf{of}\\
&\qquad\qquad\qquad\quad (l, r) \to \mathit{qsort}\ l +\!\!+ [x] +\!\!+ \mathit{qsort}\ r
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{split\_by} : t \to \mathsf{List}\ t \to \mathsf{List}\ t \times \mathsf{List}\ t\\
&\mathit{split\_by}\ \mathit{pivot}\ [\,] = ([\,], [\,])\\
&\mathit{split\_by}\ \mathit{pivot}\ (x : xs) = \mathsf{case}\ \mathit{split\_by}\ \mathit{pivot}\ xs\ \mathsf{of}\\
&\qquad\qquad\qquad\qquad\qquad (l, r) \to \mathsf{if}\ x \le \mathit{pivot}\ \mathsf{then}\ (x : l,\ h)\ \mathsf{else}\ (l,\ x : h)
\end{aligned}
$$

To type check *split_by* in the system of Hughes et al., we must choose some size i as induction variable; the natural choice is the size of the list argument (since *split_by* is defined by primitive recursion in that argument). The result depends on a dynamic test, so we can only derive a sized type with upper bounds (which are admissible by subtyping):

$$\mathit{split\_by} : \forall i.\, t \to \mathsf{List}_i\ t \to \mathsf{List}_i\ t \times \mathsf{List}_i\ t \tag{3.5}$$

Note that sizes of the result are overestimated. We would like to express a more precise relation, namely that the *sum* of the sizes of the two result lists equals the size of the argument. However, a type such as

$$\forall i\, j.\, t \to \mathsf{List}_{i+j}\ t \to \mathsf{List}_i\ t \times \mathsf{List}_j\ t$$

is *not* admissible by rule (3.4), since we cannot do induction on $i + j$. Using (3.5) as assumption for *split_by*, the type system still accepts quicksort with the type

$$\mathit{qsort} : \forall i.\, \mathsf{List}_i\ t \to \mathsf{List}_\omega\ t$$

which does not give an upper-bound for the size of the sorted list. Note, however, that due to the non-standard type semantics, the above sized type still ensures the termination of *qsort*.

Similarly, the sized type system is not well suited for algorithms over non-linear data structures such as trees (even though the theory of sized types is developed for generic algebraic data types). This is because the notion of size is always the depth of constructors and size relations must be linear; for example: a tree traversal algorithm exhibits complexity that is linear on the number of nodes but exponential on the tree depth and therefore would not be expressible. These limitations suggest that the sized type system, while guaranteeing very strong properties (termination/productivity), is also very restrictive in practice.

[11] For simplicity, we use in this example a Haskell-style syntax for list operations. We also avoid issues of *ad-hoc* polymorphism by assuming some monomorphic type t with a total order $\leq$.

The original sized type system of Hughes et al. deals with a purely denotational notion of size, but not space or time costs. In a later work, Hughes and Pareto (1999) extended the size type system with effects approximating stack and heap costs for a prototype language called Embedded ML.

Embedded ML is first-order and with a strict semantics. The model of stack and heap costs is given by an abstract machine based on the SECD (Landin 1964). Dynamic heap allocation and deallocation is done using *regions*. The standard region system of Tofte and Talpin (1997) introduces an allocation primitive

$$\mathsf{letregion}\ \rho\ \mathsf{in}\ e$$

where ρ is region variable that can used for allocations in e. After evaluation of e, region ρ is deallocated. The type and effect system of Tofte and Talpin guarantees that well-typed programs dot not access regions after deallocation.

The combination of sized types and regions allows sizes of regions to be specified at the point of allocation; overflow is prevented at compile-time by the type system. Thus, the region allocation becomes

$$\mathsf{letregion}\ \rho\#e'\ \mathsf{in}\ e$$

where e' is an expression that specifies the size of region ρ. The type judgements

$$\Gamma \vdash e : \tau\ !\ \delta; p; \phi$$

are extended with effects δ, p and ϕ: δ is the *stack effect*, p is the *put effect* and and ϕ is the *store effect*. The stack and store effect are natural numbers and approximate the maximum stack depth and heap allocations during evaluation of e. The put effect tracks allocations done in regions in the current scope.

Type checking can now ensure at compile-time the absence of space overflow. For example, consider a function that constructs a list of naturals[12]:

$$\begin{aligned} \mathit{nats}\ n\ r = \mathsf{Cons}\ n\ (\mathsf{case}\ n\ \mathsf{of}\ & \\ & 0 \rightarrow \mathsf{Nil}\ r \\ & \mid\ m+1 \rightarrow \mathit{nats}\ m\ r)\ r \end{aligned}$$

The type checker accepts *nats* with type

$$\mathit{nats} : \forall k\, r.\, \mathsf{Nat}_k \times r \rightarrow \mathsf{List}_{k+1}\ (\mathsf{Nat}_k)\ r\ \mathsf{with}\ \delta = 5k; r{+}{=}\ 3k+1$$

which specifies both stack and heap allocation as functions of the size k. The stack effect accounts for 5 stack words at each recursive invocation: one word for each

[12] Like the system of Tofte and Talpin, constructors such as Nil and Cons take an extra region argument to specify where values are to be allocated.

bound variable n, r, m, the intermediate result and the return address. The put effect specifies that region r can grow by (at most) $3k+1$ heap cells (the constants are derived from the particular operational semantics). We now see that the application

$$\mathsf{letregion}\ r\#13\ \mathsf{in}\ \mathit{length}\ (\mathit{nats}\ 4\ r)$$

is rejected by the type system because the local region r is too small for the computation of *nats*: the size k of the list is 5 (not 4) so at least $3 \times 5 + 1 = 16$ heap cells are required. To fix the program, it suffices to specify a larger size for region r. The correctness of the type system with respect to an abstract machine is presented in detail in Pareto (2000).

One first limitation of this work is that the let-region allocation is *not* sufficient to obtain bounded space behaviour in reactive systems because region lifetimes have to be nested. To deallocate values but re-use regions, Hughes and Pareto propose extending their system with *region resetting* (Birkedal et al. 1996). Furthermore, Embedded ML has no language mechanism for specifying reactive or infinite computations. Streams cannot be implemented as ordinary data types because the language has a strict semantics and is first-order.

Limitations regarding the expressive power of the recursion rule that applied to the size system alone also hold here.

Another drawback of Hughes and Pareto's approach is that it requires user annotations of both sizes *and* costs. While sizes are denotational properties that programmers can reason about in a high-level language, stack and heap costs are dependent on implementation details. Requiring the user to specify costs in type annotations (even if these are checkable by the compiler) lowers the level of software development. Even if fully automatic inference is not feasible, it would be preferable to have the user write size annotations and have the system infer costs automatically; this was left as future work in (Hughes and Pareto 1999) and not addressed in (Pareto 2000).

Chin and Khoo (2001) addressed the problem of *inferring* rather than just *checking* sized types. This system extends the prior work in two regards: first, they allow sizes to be expressed as general *Presburger constraints* (first-order logic formulae with linear arithmetic over the integers)[13] and second, by presenting an algorithm that computes a size formulae for a recursive function using an operation of "transitive closure" on constraints (Kelly et al. 1996).

[13] By contrast, type annotations in the system of Hughes and Pareto are restricted to linear size expressions. The type checking algorithm, however, generates a set of Presburger constraints for verifying the admissibility of typing.

For example, the analysis of Chin and Khoo infers the following size information for the standard list append function:

$$\begin{gathered} \mathit{append} : \mathsf{List}^m\ t \to \mathsf{List}^n\ t \to \mathsf{List}^l\ t \\ \text{s.t. } \mathbf{size}\ m \geq 0 \wedge n \geq 0 \wedge l = m + n \\ \mathbf{inv}\ 0 \leq m^+ < m \wedge n^+ = n \end{gathered}$$

The size constraint expresses the dependency between input and output list sizes, while the invariance constraint expresses properties that hold for all recursive calls (where n^+ and m^+ are the sizes of arguments in recursive calls). Invariants such as these are useful in termination analysis or in programming transformations such as partial evaluation. We will focus here on inference of size relations, since similar techniques are used for inference of invariants.

Note that the size information on the list length is more precise than what could be expressed in the system of Hughes et al.: rather than just an upper bound, the equality constraint expresses the exact result size. In general, it is possible to express lower bounds, upper bounds or equalities (simultaneous lower and upper bound).

The term language is a strict, higher-order functional notation with integers, booleans and lists. Data types are annotated with size variables and all size information is expressed by separate size constraints. Thus typing judgements take the form

$$\Gamma \vdash e :: (\tau, \phi)$$

where e is an expression, τ an annotated type and ϕ a constraint on the annotations of τ expressing the size of e.

The notion of size is specific to each data type: the size of a list is its length; the size of an integer is its value (negative sizes for negative integers); boolean values False and True have sizes 0 and 1, respectively. Assigning sizes to booleans (and other enumerated types) allows expressing control flow information in size constraints. For example, consider the function testing a list for emptiness:

$$\mathit{null}\ \mathit{xs} = \mathsf{case}\ \mathit{xs}\ \mathsf{of}\ [\,] \to \mathsf{True}\ \mid\ x : \mathit{xs}' \to \mathsf{False}$$

The sized type inferred for *null* is

$$\mathit{null} :: (\mathsf{List}^n\ a \to \mathsf{Bool}^c,\ (n = 0 \wedge c = 1) \vee (n > 0 \wedge c = 0))$$

where the size c of the boolean result encodes which branch of the conditional was taken.

Unlike the system of Hughes and Pareto, the typing rule for recursive functions in Chin and Khoo's system does not impose a well-founded order on sizes. Again omitting type generalisation for simplicity, the type rule is:

$$\frac{\Gamma \cup \{x :: (\tau_1,\, \phi_1)\} \vdash e_1 :: (\tau_1,\, \phi_1) \qquad \Gamma \cup \{x :: (\tau_1, \phi_1)\} \vdash e_2 :: (\tau_2,\, \phi_2)}{\Gamma \vdash \mathsf{letrec}\ x = e_1\ \mathsf{in}\ e_2 :: (\tau_2,\, \phi_2)} \tag{3.6}$$

Re-visiting our erroneous length function example, rule (3.6) allows us to type it as

$$\mathit{wronglen} :: (\mathsf{List}^i\ a \to \mathsf{Int}^j,\ i \geq 0 \wedge j \geq 0)$$

which expresses approximate information about the function semantics (namely, that both the argument and result must have non-negative sizes). A more precise information about *wronglen* is expressed by the sized type

$$\mathit{wronglen} :: (\mathsf{List}^i\ a \to \mathsf{Int}^j,\ i = 0 \wedge j = 0)$$

that is, *wronglen* is only defined for the empty list. Both sized types are admissible by rule (3.6).

The treatment of recursion in the system of Chin and Khoo corresponds to a distinct objective to that of Hughes and Pareto: rule (3.4) guarantees a *liveness* property (termination/productivity) whereas rule (3.6) guarantees a *safety* property (approximation of the dynamic sizes of values).

The type rules for expressions and non-recursive functions in Chin and Khoo's system are syntax directed, so that size type inference can be done by synthesising constraints from sub-expressions. However, rule (3.6) for letrec is not syntax directed since the size constraint for the result appears in the hypothesis.

To compute a size constraint for a recursive function

$$\mathsf{letrec}\ f = \lambda x.\, M\ \mathsf{in}\ N$$

Chin and Khoo first compute a constraint expressing the size-change between two successive recursive iterations:

$$\lambda f.\, \lambda x.\, M$$

They then employ an algorithm to approximate the *transitive closure* of a Presburger constraint (Kelly et al. 1996). One further difficulty is that the transitive closure might not be expressible as a Presburger formula and the algorithm sometimes yields lower-bound approximations. This is inadequate for the type rule (3.6), so some post-processing steps are employed to obtain a safe upper bound. These computations are implemented using the Omega Calculator (Pugh 1992).

Chin and Khoo formulate the soundness of their size analysis with respect a standard higher-order denotational semantics. The proof, however, has one important technical flaw: it relies the existence of a constraint $\mathcal{S}(v :: \tau)$ describing the exact size of a value v of annotated type τ. While this is valid for zero-order values, it fails to hold for functional values because of the lattice of constraints is an incomplete partial order. We will re-visit this issue in Section 5.4 where we present a revised proof for the first-order case.

List constructors have specialised typing rules instead of being treated as constants as in the sized type systems (Reistad and Gifford 1994, Hughes et al. 1996); this is required to derive a type for lists where the head has different size than the tail.

$$\frac{\Gamma \vdash e_1 :: (\tau_1,\, \phi_1) \qquad \Gamma \vdash e_2 :: (\mathsf{List}^m\, \tau_2,\, \phi_2)}{\Gamma \vdash (e_1 : e_2) :: (\mathsf{List}^n\, \tau,\, n = m + 1 \wedge \phi_1 \wedge \phi_2 \wedge (\tau = \tau_1 \vee \tau = \tau_2))} \tag{3.7}$$

In rule (3.7) the type τ is constrained to be identical to τ_1 and τ_2 except for "fresh" size annotations; the equations $\tau = \tau_1$ and $\tau = \tau_2$ specify the equality constraints between size annotations in two types; the size constraint for the application specifies the length of the result list and size elements.

One limitation of the type system is that while rule (3.7) can be used to infer size relations on the list lengths, it often fails to infer sizes of values *inside* lists. Consider for example, the *tail* function on a list of integers:

$$\mathit{tail}\ \overbrace{(\underbrace{x}_{\mathsf{Int}^k} : \underbrace{xs}_{\mathsf{List}^m\ \mathsf{Int}^j})}^{\mathsf{List}^n\ \mathsf{Int}^i} = \overbrace{xs}^{\mathsf{List}^m\ \mathsf{Int}^j}$$

Rule (3.7) generates the size constraint

$$n = m + 1 \wedge (\mathsf{Int}^i = \mathsf{Int}^k \vee \mathsf{Int}^i = \mathsf{Int}^j) \iff n = m + 1 \wedge (i = k \vee i = j)$$

where m, k and j are "fresh" size variables; to obtain the type for the function, the constraint is simplified by existentially quantifying variable k that does not occur in the argument or result type; this yields the sized type

$$\mathit{tail} :: (\mathsf{List}^n\ \mathsf{Int}^i \to \mathsf{List}^m\ \mathsf{Int}^j,\ n = 1 + m)$$

where no size information is obtained for elements inside the list.

In a subsequent work Chin et al. (2003) propose an extension to the sized type system with collection constraints to address this problem. However, the extended constraints fall outside the capabilities of a Presburger solver; the cited paper does not address the issue of solving these combined constraints.

Another limitation is that the type system is not type polymorphic since no size information is captured for type variables. The system is still able to obtain good size information for monomorphic instances. For example, the first tuple projection can be typed *fst* : $(\mathsf{Int}^i \times \mathsf{Int}^j \to \mathsf{Int}^k,\ k = i)$ but no size information is obtained for the polymorphic version *fst* : $(\forall a \forall b.\, a \times b \to b,\ \mathit{True})$.

The size type system of Chin and Khoo allows higher-order functions, but no size relations are inferred from uses of functional arguments. For example, no sizes are inferred for the usual *compose* function $\lambda f.\, \lambda g.\, \lambda x.\, f\ (g\ x)$. Moreover, since the type system does not capture sizes for polymorphic functions, there is no analogue of a "principal size type" to be inferred in such cases.

3.4 Dependent types

The defining characteristic of dependent type systems is the possibility of parameterising types over values. Dependent type systems generalise the function type $A \to B$ to the *dependent product* $\Pi x : A.\, B$ where the type B of the co-domain is allowed to vary with x; the simple function type is obtained as an instance where x does not occur in B.

Restricted forms of type dependency have long been used in programming languages. For example, the Pascal array type depends on its size; and the types of arguments of the C-language `printf` depend on its first argument (the format string). Dependent type systems are formal basis for reasoning about such notions.

Following the Curry-Howard correspondence (Girard et al. 1989), dependent types allow expressing both *propositions* and *computational* (data) types in a single framework; therefore dependent type theories can form the basis of proof assistants, e.g. *Coq* (Coq 2006) and program verifiers, e.g. *Lego* (Luo and Pollack 1992).

More recently, there has been an increase of research in functional programming languages incorporating dependent types, e.g. *Dependent ML* (Xi 1998), *Cayenne* (Augustsson 1998), *Agda* (Coquand and Coquand 1999) and *Epigram* (McBride and McKinna 2004). This is motivated by the desire to express more refined program properties using types than is possible with the standard polymorphic type systems. In fact, some extensions of the Haskell type system implemented in GHC, e.g. type classes with functional dependencies (Jones 2000) and generalised algebraic data types (Jones et al. 2006) allow simulating some of the expressive power of dependent types (McBride 2002, Apple and Weimer 2007).

Dependent ML (DML) is a conservative extension of the ML language with dependent types (Xi 1998, Xi and Pfenning 1999). The motivation for DML was to extend a realistic programming language with dependent types whist retaining both decidability of type checking and a low overhead of type annotations. This is achieved by separating arbitrary ML terms (where general recursion is allowed and whose equivalence is therefore undecidable) from the *indices* allowed in types (taken from some decidable constraint domain).

Computation on DML type indices is restricted to constraint normalisation; this allows reducing the type checking of DML programs to constraint solving in the underlying domain. The constraint domain of natural indices with addition allows capturing size invariants of data structures; deciding the equivalence of the DML types with such indices can then be reduced to checking equivalence of Presburger constraints, e.g. using the Omega calculator (Pugh 1992). Xi (1998) presented applications of DML types with natural indices to program error detection and optimisations, e.g. elimination of array bounds check and dead-code.

Dependent types in DML are introduced by *refining* a standard data type declaration. For example, a canonical declaration for a list data type

```
datatype 'a = nil | cons of 'a * a' list
```

can be refined with a natural length measure by the declaration:

```
typeref 'a list of nat with
   nil <| 'a list(0)
 | cons <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)
```

This refinement assigns a type with length zero for `nil` and a type for `cons` that increases the length by one; the notation {`n:nat`} is the concrete syntax for introducing a dependent product $\Pi n : \mathsf{nat}$. Size properties regarding lists can then be expressed by dependent type annotations; for example, the size relation for the list append function is expressed by the type

```
append <| {m:nat}{n:nat} 'a list(m) * 'a list(n) -> 'a list(m+n)
```

and the DML type checker can verify that this size relation holds for the canonical recursive definition of append.

For size relations that cannot be expressed exactly, DML allows the use of *dependent sum types*. For example, the higher-order *filter* function computes a sub-list of elements verifying some predicate; since the length of result depends on the predicate it cannot be specified exactly; however, an upper-bound can be specified by the type

```
filter <| ('a -> bool) * {n:nat} 'a list(n)
                                   -> [m:nat | m<=n] 'a list(m)
```

where `[m:nat | m<=n]` is a *dependent sum* that constraints the result list length m to be at most the length n of the original.

DML with integer indices allows expressing properties similar to the sized type systems (Reistad and Gifford 1994, Hughes et al. 1996, Chin and Khoo 2001). The main distinctions between the two approaches are: DML indices are user-definable for each data type whereas the notion of “size” in the sized type systems above is rigid; there is no implicit subtyping relation for size coercion in DML (instead, relevant functions must be annotated with dependent sum types); and finally, the DML type checker can *verify* user-annotated size relations but not *infer* them as in (Reistad and Gifford 1994, Chin and Khoo 2001).

Grobauer (2001) presented a method for automatically deriving cost recurrences from first-order DML programs. The main contribution is the use of indices in DML types as data sizes for expressing the recurrences. This allows the user to specify more precise size measures for data, e.g. nested lists or trees. The cost model is asymptotic (e.g. the number of function calls or some other primitive operation). This work focuses on extracting cost recurrences but not on obtaining *solutions* to the cost equations. Except in very simple cases, obtaining closed form solutions requires human intervention. For example, a function merging two lists in order (part of a merge sort example)

```
fun merge l = case l of
   (nil, l2) => l2
   (l1, nil) => l1
   (cons(h1,t1),cons(h2,t2)) =>
       if h1<h2 then cons(h1, merge(t1,l2))
                else cons(h2, merge(l1,t2))
with merge <| {n1:nat}{n2:nat} list(n1)*list(n2) -> list(n1+n2)
```

yields the following cost recurrence (braces represent possibly-guarded maximum between alternatives):

$$\mathit{merge}^c\ n_1\ n_2 = \begin{cases} n_1 = 0 \mapsto 0 \\ n_2 = 0 \mapsto 0 \\ n_1 > 0 \wedge n_2 > 0 \mapsto 1 + \begin{cases} \mathit{merge}^c\ (n_1 - 1)\ n_2 \\ \mathit{merge}^c\ n_1\ (n_2 - 1) \end{cases} \end{cases} \tag{3.8}$$

It is immediate that the cost recurrence mimics the recursive structure of the original function. Even using computer algebra systems such as *Maple* or *Mathematica*, some human intervention is required to convert a recurrence such as (3.8) into the closed form expression $\mathit{merge}^c\ n_1\ n_2 = \min(n_1, n_2)$.

Crary and Weirich (2000) used a system based on *proof-carrying code* (Necula

1997) to perform verification of resources bounds. This system is based on a intermediate compiler language called *LXres* that allows expressing resource properties in types by exposing a “virtual clock” representing some available resource (e.g. time). Resource properties can then verified by the type checker. To deal with variable-time procedures, they employ a technique of encoding static type-level representations of data using using sum and inductive kinds; this simulates type dependency while allowing a simpler theory and type checker.

Costs can be expressed as primitive-recursive functions over the static data representations (so that type checking remains decidable). These must be provided by the user: the system allows *verifying* resource bounds, but makes no attempt to *infer* them.

Brady and Hammond (2006) employed a dependently-typed language similar to *Epigram* to encode and verify size properties of functional programs. Their approach generalises the previous examples of sized lists in DML by introducing a dependent type Size that pairs a type indexed by a natural size and a predicate (itself represented as an dependent type). A term $\mathsf{size}\ v\ p : \mathsf{Size}\ A\ P$ pairs a value v of indexed type $A\ n$ and a proof p that v respects a size property P.

Brady and Hammond applied this framework to express size relations of functions on lists, including an example similar to the *split_by* function of Section 3.3. They also extend the technique to capture size relations for higher-order functions by associating size predicates and functions with higher-order arguments. The authors illustrate the technique with the higher-order functions such *twice*, *map* and *fold*.

A first limitation of this work is that it considers only verifying sizes expressed as dependent types. The elaboration of a simply typed program in Haskell or ML into a dependently typed version with size annotations is left to the user (particularly guessing size relations of functions). The extent to which this step can be automated is not addressed.

Secondly, this work uses dependent types for expressing size information but not time or space costs. Although the authors mention that the technique should be extendible to other metrics such as heap, stack or time usage, we remark that such extension is not likely to be straightforward because it requires reasoning also about *intentional* properties of evaluation (e.g. cost) rather than just *denotational* ones (e.g. size).

Danielsson (2008) has also used a dependently-typed language for expressing complexity analysis of functional programs. This work focuses on expressing costs rather than sizes by encapsulating values in a cost monad (Wadler 1993) parameterised by the number of computation steps: $\mathsf{Thunk}\ n\ a$ is the type of a computation that evaluates to an a in n steps. The unit and bind operations for the thunk monad are:

$$\begin{aligned} \mathit{return} &: a \to \mathsf{Thunk}\ 0\ a \\ \gg\!= &: \mathsf{Thunk}\ m\ a \to (a \to \mathsf{Thunk}\ n\ b) \to \mathsf{Thunk}\ (m+n)\ b \end{aligned}$$

The monadic unit injects a value into the cost monad with zero cost while the bind combines costs from two computations. Any atomic costs must be explicitly introduced using "tick" annotations in the program; each *tick* adds one unit of cost:

$$\mathit{tick} : \mathsf{Thunk}\ n\ a \to \mathsf{Thunk}\ (1+n)\ a$$

Note that the Thunk type is dependent on the natural n and that both the monadic operations and *tick* have dependent types.

These basic combinators form a library implemented in the dependently typed language *Agda* and allow a programmer to specify machine-checkable complexity proofs; for example, assuming a dependent type for lists annotated with their length, and assigning a unit cost to each lambda-abstraction, we can type check a list concatenation function annotated with a linear cost on the first argument:

$$\begin{aligned} (+\!\!+) &: \mathsf{List}\ m\ a \to \mathsf{List}\ n\ a \to \mathsf{Thunk}\ (1 + 2 * m)\ (\mathsf{List}\ (m+n)\ a) \\ [\,]\ +\!\!+\ ys &= \mathit{tick}\ (\mathit{return}\ ys) \\ (x : xs)\ +\!\!+\ ys &= \mathit{tick}\ (xs\ +\!\!+\ ys \gg\!= \lambda t \to \mathit{tick}\ (\mathit{return}\ (x : t))) \end{aligned}$$

The use of a dependently-typed cost monad allows expressing quite precise cost information, e.g. it can be used to reason about the complexity of *lazy* evaluation by explicitly embedding Thunk types into data structures.

However, it requires insightful annotations by the user and a considerable knowledge of dependent type systems. For example, to type check the concatenation example above requires providing a lemma for the arithmetic equality $1 + ((1 + 2 * m) + (1 + 0)) = 1 + 2 * (m + 1)$. Non-trivial programs also require the introduction of auxiliary operators, e.g. to "waste" costs and ensure that the two branches of a conditional admit the same type[14].

The cost model used is quite abstract: it counts number of "steps" specified by the number of ticks annotated in the code. Presumably the technique could be extended to a model of cost based on an abstract machine, e.g. as in (Hughes and Pareto 1999).

[14] This is analogous to the subeffecting allowed in effects systems for time (Reistad and Gifford 1994) expect that the latter is implicit.

Finally, the system allows only *checking* cost bounds but does not aid in obtaining the cost bounds in the first place.

3.5 Amortised cost analysis

Amortised complexity analysis aims at obtaining bounds for the cost of a sequence of operations (Tarjan 1985, Okasaki 1998); it is sometimes possible to obtain better worst-case bounds by amortisation than by reasoning about the costs of individual operations. For example, it might be possible to obtain a worst-case bound of $O(n)$ for a sequence of n operations even if some of the individual operations cost more than $O(1)$.

The "physicist method" for deriving amortised bounds starts by assigning a non-negative *potential* function to data. The *amortised cost* of an operation is then defined as the sum of the actual cost (e.g. time cost or heap cells allocated) plus the difference in potential incurred by the operation. The key idea is to choose the potential functions so as to facilitate computing the amortised cost, e.g. in such a way as to make the amortised costs constant. Provided the potential is always non-negative and initially zero, the accumulated amortised costs will be an upper-bound on the accumulated actual costs (Okasaki 1998).

Hofmann and Jost (2003) proposed a type-based analysis for heap space usage using amortisation. Instead of extending type judgements with effects as in (Dornic et al. 1992, Reistad and Gifford 1994, Hughes and Pareto 1999), the analysis of Hoffman and Jost is based on annotating data types with weights representing the relative contribution of parts of a data structure to the overall heap usage (the *potential* associated with the data structure).

The language under analysis is a first order functional notation with a strict semantics and algebraic data types including sums, products, booleans and lists. There are two kinds of pattern-matching deconstructors for heap-allocated values: a deallocating match and non-deallocating match'. The heap cost is defined by a big-step operational semantics instrumented with the size of a free list of heap cells; the free list reduces at each constructor application and grows at each match (but not at match').

The augmented typing judgements take the form Γ, $k \vdash e : A$, k' where Γ are the type assumptions, e is an expression, A is an annotated type and k, k' are non-negative rational numbers representing the available potential before and after the evaluation of e. The annotations in A together with k and k' give both an upper bound on the initial heap space for evaluation of e and a lower bound on the available heap space after evaluation. For example, the judgement

$$x : \mathsf{L}(\mathsf{L}(\mathsf{B}, 1), 2),\ 3 \vdash e : \mathsf{L}(\mathsf{B}, 4),\ 5$$

informally says that if x is a list of lists of booleans then e is a list of booleans; furthermore, if $x = [l_1, \ldots, l_n]$ then a free list of size $3 + 2n + 1 \sum_i |l_i|$ is sufficient to evaluate e; and if e evaluates to a list $[b_1, \ldots, b_m]$ of length m, the resulting free list will have size at least $5 + 4m$.

From this example we can see that type annotations play a very different role here than in the sized type systems: in the system of Hoffman and Jost an annotation represents not a *size*, but the *coefficient* of the heap cost incurred by a part of a data structure. The upper bound on the initial free list is a function of the (unknown) sizes of the input. Note also that the lower bound on the final free list size is a function of the (unknown) size of the output and that no input/output size relation is obtained.

The type system of Hofmann and Jost performs an amortised analysis of the size of the free list: the coefficients in types represent the *potential* associated with the data structures; the typing rules constrain the annotations so that the amortised costs for each expression are properly accounted. For example, the typing rules for constructing and deconstructing a list node are:[15]

$$\frac{n \geq \mathsf{SIZE}(A \otimes \mathsf{L}(A, k)) + k + n'}{\Gamma, x_h : A, x_t : \mathsf{L}(A, k), n \vdash \mathsf{cons}(x_h, x_t) : \mathsf{L}(A, k), n'} \tag{3.9}$$

$$\frac{\begin{array}{c}\Gamma, n \vdash e_1 : C, n' \\ \Gamma, x_h : A, x_t : \mathsf{L}(A, k), n + \mathsf{SIZE}(A \otimes \mathsf{L}(A, k)) + k \vdash e_2 : C, n'\end{array}}{\Gamma, x : \mathsf{L}(A, k), n \vdash \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \\ \mid \mathsf{cons}(x_h, x_t) \Rightarrow e_2 \end{array} : C, n'} \tag{3.10}$$

Rule (3.9) specifies that the available potential n must be at least the amortised cost of cons, that is, the actual heap cells used (given by the SIZE function) plus the potential k associated with the list elements (because the list length is increased by one). Dually, rule (3.10) specifies that the available potential at the cons alternative increases by the amortised cost (because match does deallocation).

Hofmann and Jost presented an algorithm that automatically infers the type annotations. Their technique associates each program P with a system of linear inequalities $\mathcal{L}(P)$ such that the valid annotated type derivations for P correspond to the admissible solutions of $\mathcal{L}(P)$; these solutions can be obtained by a standard linear programming solvers.

[15] Following Hofmann and Jost (2003) and without loss of generality, we present the type rules for expressions in let normal form.

The worst-case theoretical complexity for solving linear programs is polynomial; the variants of the simplex algorithm used in solver implementations, although exponential in the worst-case, are quite efficient in practice. This compares favourably with the sized type systems (Hughes et al. 1996, Hughes and Pareto 1999, Chin and Khoo 2001) where type checking alone requires checking validity of Presburger constraints with doubly-exponential worst-case time.

Since annotations represent coefficients of the potential function, the system can only derive heap bounds that are linear on the sizes of data structures. However, since the language implements deallocation using destructive matching, it is still expressive enough to obtain heap costs for many list processing functions, including insertion algorithms such as insertion sort and quicksort.[16] Unlike the sized type analysis of Hughes and Pareto (1999), the amortised analysis deals with the irregular divide-and-conquer recursions by "splitting" the potentials between the two recursive calls. Hofmann and Jost also present good results for a binary tree traversal and report successfully analysis of other textbook examples.

One limitation of the analysis of Hofmann and Jost is that the inferred type annotations are sometimes not sufficiently polymorphic because every use of a function shares the same potentials. Consider the identity function $f : \mathsf{L}(\mathsf{B}) \to \mathsf{L}(\mathsf{B})$ on a list of booleans; if a particular use requires the annotation $f : \mathsf{L}(\mathsf{B}, 5), 3 \to \mathsf{L}(\mathsf{B}, 5), 3$ then it not possible to apply f to an argument of type $\mathsf{L}(\mathsf{B}, 0)$. The authors suggest that this can be relaxed by conducting separate analysis for each use of f. However, this implies that is not possible to analyse functions separately from their use, i.e. the analysis is not fully modular.

Hofmann and Jost have considered heap usage but not time or stack usage. Time could, in principle, be treated similarly to heap, by simply recording the number of execution steps instead of the size of a free list. The only difference is the absence of a deallocation mechanism for time costs.

Extending the amortised analysis for stack usage is less straightforward. One technical problem is that a realistic model for stack must employ a small-step rather than a big-step semantics as used in (Hofmann and Jost 2003). Another concern is that the bounds expressible by the amortised analysis are linear on the size of data structures (the total number of elements). While this is generally a good match for obtaining heap bounds, for example, it will yield coarse stack bounds for a tree search algorithm whose worst-case complexity is linear on the depth of the tree. A recently submitted PhD thesis investigates the extension of amortised analysis

[16] The sorting algorithms exhibit linear space or even constant bounds by reusing the heap associated with the input list for constructing the sorted list, i.e. they destroy the original list.

to stack costs; the definition of potential is modified to account the depth of data structures (Campbell 2008).

3.6 Other related work

Hofmann (2000) has proposed the use of a linear typing discipline for ensuring that data structures are used in a single-threaded way and can therefore be update in-place. Hofmann further shows that first-order functional programs that admit a linear type in this system can be translated into C-language programs with bounded space behaviour by construction: there are no uses of `malloc()` because all data structures are updated in-place; thus dynamic memory requirements are bounded by the usage of initial data. Of course, this guarantee applies to heap but not to stack.

Some early works on complexity theory have studied complexity bounds of Turing-incomplete languages. Meyer and Ritchie (1967) studied the complexity of bounded loop programs; such programs cannot implement all computable functions but can implement the first-order primitive-recursive functions on naturals (Lewis and Papadimitriou 1981, chapter 5). The complexity bounds are expressed using a family of primitive-recursive functions indexed by the depth of loop nesting and the number of instructions in the program. However, the bounds are rather coarse, e.g. a program with a single loop is bounded by a linear function but a program with two nested loops is bounded by an exponential.

Turner (1995, 2004) proposed a discipline for *strong* functional programming, that is, where program termination is guaranteed by construction. The principal objective is the simpler equational theory resulting from the absence of a “bottom” value associated with partiality. Unlike approaches based on constructive type theory, Turner proposes an *elementary* discipline that could be used at an introductory programming level; he restricts a pure functional language such Miranda or Haskell by:

1. requiring all case-analysis definitions to be exhaustive;
2. extending all built-in operations to be total (e.g. arithmetic);
3. requiring arguments of recursive calls to be structural sub-components of the formal parameters;
4. requiring recursive data types to be *covariant* (that is, recursion on the left of the arrow type constructor is disallowed).

The resulting programming language is not Turing-complete, but is expressive enough to encode higher-order primitive recursive functions over naturals and other inductive types. To express non-terminating interactions (e.g. an operating system), Turner proposes separating the recursive data types which must be finite (e.g. naturals and lists) from corecursive ones which are infinite (e.g. streams). Corecursive definitions must be guarded by co-constructors; this is sufficient to ensure that codata values are productive.[17]

Turner argues that the restriction to primitive recursive definitions captures most useful computable functions. However, algorithms must sometimes be re-written with worse time or space complexity than an equivalent general-recursive formulation; this is undesirable for resource-constrained systems. In any case, we remark that the restriction to a total programming language does *not*, by itself, guarantee resource bounds, except in the naive extensional sense referred in Chapter 1.

A prerequisite for all cost analysis is to choose a *model of costs*. Most of the previous works (Le Métayer 1988, Rosendahl 1989, Wadler 1988, Bjerner and Holmström 1989, Dornic et al. 1992, Reistad and Gifford 1994, Vasconcelos and Hammond 2004) chose to count the number of function calls (or the corresponding formal notion of β-reductions in the lambda-calculus). This metric has the advantage of being easily understood by relation with a naive equation rewrite semantics for an applicative language. However, this is not likely to give accurate time predictions because each β-reduction can require different amounts of execution time.

On the theoretical side, Dal Lago and Martini (2005) have argued against using the number of β-reductions as a cost model for the lambda-calculus. They proposed a model where the cost of a reduction $M \to N$ is proportional to the difference $|N| - |M|$ between the sizes of *redex* and *reduct* and prove that it satisfies a polynomial-invariance result, i.e. that it can be simulated by a Turing machine within a polynomial-time bound overhead and vice-versa (unlike the cost model based on β-reductions).

Hope and Hutton (2006) proposed counting the reduction steps of an abstract machine. Such a model stands half-way between the very abstract measure (number of β-reduction) and measuring real-time for a concrete implementation. Hope and Hutton follow the methodology of Ager et al. (2003*b*) and Danvy (2003): starting from a denotational evaluator for the language, they apply a sequence of meaning-preserving program transformations to obtain an abstract machine interpreter; this

[17] This is similar to the size type system of Hughes et al. (1996); the latter, however, ensures termination and productivity by a *semantic* properties of sizes rather than *syntactical* restrictions.

interpreter is then straightforwardly extended with a step-counter; finally, the program transformations are reversed to get back a cost-instrumented denotational evaluator. The principal strength of this approach is that the program transformations are calculated, thus giving a constructive methodology for reasoning about costs of an implementation at the source level.

In his PhD thesis, Bakewell (2001) studied operational theories for reasoning about space usage of programming language evaluators. He defined a graph rewrite system to formally specify the operational semantics of distinct language evaluators, including implementation techniques of lazy evaluation and garbage collection. Bakewell is then able to compare the space behaviour of evaluators by means of simulation proofs—for example, he is able to prove that one evaluator is not leakier than another. He also classifies space leaks according to the behaviour that exposes them and provides a search procedure to find proofs of space leaks.

A more pragmatic approach to determining time and space usage is simply to profile execution runs. One of the difficulties in implementing a profiler for a language with higher-order functions and lazy evaluation is deciding how to assign costs to the source program. Samson and Jones (1995) addressed this problem in the Haskell time profiler by labeling the program with *cost centers*, either by hand or automatically (e.g. one cost center for each top-level definition). This is approach is employed in the Glasgow Haskell Compiler (GHC).

Runciman and Wakeling (1993) developed a heap space profiler for *Lazy ML* (a dialect of ML with lazy evaluation) together with a set of tools for visualing results. This was quickly adapted to other lazy functional languages and is currently a distributed as part of the GHC profiler.

While not providing the strong guarantees of static analysis, the Haskell time and space profilers can be very useful in improving the performance of lazy functional programs. In particular, it has proved effective in detecting the causes of "space leaks" in Haskell programs; these can then be addressed e.g. by program transformation or by adding strictness annotations. Runciman and Wakeling demonstrate a reduction of heap residency of two orders of magnitude in a simple example (a propositional formula simplifier). Hudak et al. (2007) report that time profiling the GHC compiler (itself written in Haskell) has allowed improving its running time by a factor of two.

Bonenfant et al. (2007) conducted worst-case execution time (WCET) analysis to obtain bounds on real-time costs for a subset of the abstract machine instructions of *Hume*, a functionally-inspired research language for resource-sensitive systems (see Chapter 4). Their approach is to translate the abstract machine instructions into C and use a C compiler to obtain machine code; they then employ *aiT*, a commercial tool for static WCET analysis of machine code blocks (Ferdinand et al. 1999, 2001). Unlike approaches based on experimental tests, the *aiT* tool uses abstract interpretation to model cache and pipeline states of specific microprocessors and is capable of obtaining *guaranteed* worst-case time bounds. Bonenfant et al. applied this tool to derive WCET costs of compiled code for a Renesas M32C/85 micro-controller, compared the results with experimental timings and report a close match with the analysis bounds.

Chapter 4

Core Hume

Hume is a research language aimed at applications requiring bounded time and space behaviour, such as real-time embedded systems. In this chapter we review the Hume language and describe a subset, called *Core Hume*, that will be the subject of the size and space analyses developed in this thesis.

The development of this chapter is as follows: in Section 4.1 we present the Hume language in an informal way; the remaining sections we define the Core Hume subset: the abstract syntax (Section 4.2.1); the type discipline (Section 4.2.2); and a denotational semantics (Section 4.2.3); Section 4.4 concludes with some example programs.

4.1 The Hume language

Hume is a research language aimed at applications requiring bounded time and space behaviour, such as real-time embedded systems (Hammond et al. 2007). The research motivation for Hume is to explore language design and static analyses for providing as high a level of programming expressiveness as possible, whilst retaining the predictability of resource usage required by real-time embedded systems.

Hume combines ideas from finite-state machines and functional programming to separate the *coordination* and *computation* aspects of embedded system programming: a Hume program defines a finite number of communicating processes; each process is a purely functional mapping of inputs to outputs. The aim for this layered approach is to facilitate combining the high expressiveness of functional programming (e.g. automatic memory management, user defined data types, pattern matching and recursion, strongly-typed and higher-order functions) with bounded time and space behaviour required in embedded systems.

4.1.1 Declaration, expression and coordination

Hume programs are structured in three layers: an outermost static *declaration* layer for defining data types, exceptions, streams, etc. to be used in the dynamic layers; the innermost pure functional *expression* layer used for defining functions and values; and the middle *coordination* layer used for defining a finite number of communicating processes.

Coordination layer

The basic unit of coordination is the *box*: a process with a finite number of inputs and outputs. The behaviour of each box is specified by a set of pattern matching rules that map input values to output values. Boxes are re-entrant, that is, they execute an indefinite event loop of waiting for inputs and producing outputs. The following box specifies a one-bit inverter:

```
box invert
in (x :: word 1)
out (y :: word 1)
match
  0 -> 1
| 1 -> 0 ;
```

Each box has a fixed set of named inputs and outputs (in the example above, `invert.x` is the single input and `invert.y` is the single output). Input and output are typed (in the example, as 1-bit words). The mapping between inputs and outputs is specified by a set of rules. In the above example, rules match literal inputs and produce literal outputs; in general, rules can bind variables using patterns and evaluate expressions. For example, the bit inverter example could also be written as:

```
box invert
in (x :: word 1)
out (y :: word 1)
match
  x -> 1-x ;
```

Expression layer

The expression layer of Hume is a pure functional notation similar to a pure subset of Standard ML or Haskell. The expression language is strongly-typed and includes user-defined data types and recursive, polymorphic and higher-order functions. To facilitate the development of static cost analysis the expression language has a strict semantics (i.e. call-by-value).

Hume includes a rich set of primitive data types to ease interfacing with hardware sensors and devices: booleans, characters, integers, bit vectors (words) and floats. Primitive types specify the data size representation explicitly, e.g. `word 1` is a 1-bit word; `int 16` is a 16-bit signed integer; `float 64` is a 64-bit floating point number, etc. The compiler should choose the implementation that best suits each data size.

The compound types include strings of characters, vectors of a fixed size, tuples, lists and user-defined data types. The latter follows the sums-of-products style of data declaration in Standard ML or Haskell: each product is tagged with a constructor label and alternatives are separated by a vertical bar. For example, the following defines a data type for binary trees where each inner node is decorated with a label of some arbitrary type:

```
data Tree a = Leaf | Branch a (Tree a) (Tree a)
```

Constructors build data values when used in expressions and decompose them when used in patterns; in the later case variables are bound to the constructors arguments. For example, the following function computes the number of inner nodes of a tree:

```
size :: Tree a -> int 32
size Leaf = 0
size (Branch x l r) = 1 + size l + size r
```

Patterns can be used in either function definitions or case expressions and may be nested and non-exhaustive. Pattern matching is done in top-to-bottom order so that overlapping rules are tried in order.

4.1.2 Communication and synchronisation

Boxes communicate with each other and the environment by single-buffered, point-to-point links called *wires*. The use of buffering decouples the writer and reader boxes, improving available concurrency. It also allows connecting the output of a box into one of its own inputs, thereby creating an explicit feedback loop; this technique can be used to represent state passed from one box iteration to the next without compromising the purely functional expression semantics.

The use of a single stage buffer aids in guaranteeing that space and time costs for communication can be bounded at compile time. Multiple stage buffering can be implemented using multiple boxes or in the expression layer, e.g. using a list data type.

Access to wires is mutually-exclusive and provides the only form of synchronisation between boxes. Reading from a wire *consumes* the value, making it unavailable for other readers. Conversely, a box that attempts to write on a non-empty wire becomes *blocked*; the blocked box is resumed only when all required output wires are available.

The Hume coordination layer semantics is deterministic, that is, two executions from the same initial state produce the same observable behaviour. To ensure determinacy boxes are scheduled in a round-robin fashion, following the order of program declarations. A box is scheduled only when all required inputs are available and executes as an atomic, non-preemptable task to produce the output values. Therefore, the only observable concurrency is the interleaving of box executions.

Synchronous programming

Hume boxes can be used to implement synchronous reactive systems. As an example, consider a simple low-pass digital filter of a stream of floating-point samples $x(n)$ that produces a filtered response $y(n)$ satisfying the equation $y(n) = \frac{1}{2}(x(n) + x(n-1))$.

```
box filter
in (x0 :: float 32, x1::float 32)
out (y0 :: float 32, x1'::float 32)
match
  (x0,x1) -> (0.5*(x0+x1), x0);

wire filter.x1 filter.x1' initially 0.0;
```

The input and output ports of the filter box are `filter.x0` and `filter.y0`, respectively. Note that the value of the previous sample is written to `filter.x1'` and wired in a feedback loop to `filter.x1`; the initial value of needed in the first iteration is (arbitrarily) set to zero.

At any given iteration only four floating point values are stored in the input and output wires; therefore the sizes of communication wires are bounded. It is also straightforward to bound time and space costs for the arithmetic expressions used in the box; therefore the total time and space resources for the filter can be statically bounded.

Asynchronous programming

Hume is not restricted to synchronous programming, i.e. there is no assumption that inputs to boxes must arrive simultaneously. The coordination layer allows boxes to react to partial inputs by using special *asynchronous patterns* in rules: the pattern "*" ignores the input (and does not consume it); the pattern "_*" ignores the input (but still consumes it if it is available). Dually, no output is produced by using "*" on the right-hand side of a box rule.

For example, the following box odds filters a stream of integers discarding all even values.

```
box odds
in (n :: int 32)
out (n' :: int 32)
match
  n -> if n%2==0 then * else n
```

The right-hand side of the rule in odds returns "*" if the input is an even number, thus producing no output. Consequently, the stream of output values will not be in synchrony with the input stream.

As another example, the following merge box writes one of its two inputs to the output, thus merging two input streams into a single one. The *-pattern allows the rule to match when only one of the values is present.

```
box merge
in (x::int 32, y::int 32)
out (xy::int 32)
match
  (x, *) -> x
| (*, y) -> y
```

Note that the merge box biases the first input stream because rule matching is done in top to bottom order: when both inputs are available, the first rule takes precedence and the value of x is output. This means that the y input can be ignored for an arbitrarily long time. To avoid this behaviour Hume allows the programmer to specify *fair* matching for box rules.

```
box fairmerge
in (x::int 32, y::int 32)
out (xy::int 32)
fair
  (x, *) -> x
| (*, y) -> y
```

Fair matching in Hume means *pattern fairness*: all matching rules must be selected equally often given an infinite stream of matching inputs.[1] Note that channel fairness is *not* enforced. It is possible (and sometimes desirable) to treat inputs unfairly. It is up to the programmer to ensure that inputs are treated fairly if required.

4.1.3 Exception raising and handling

Exceptions can be *raised* during expression evaluation either explicitly by `raise` or as a result of some illegal operation (e.g. division by zero). However, exceptions can only be *handled* in the coordination layer. Exception handlers can be declared for a box that will treat exceptions raised in any right-hand side of the box rules.

For example, the following box handles the numeric `Overflow` exception by producing a null result.

```
box average
in (x :: float 32, y :: float 32)
out (z :: float 32)
handles Overflow
match
  (x,y) -> ((x+y)/2)
handle
  Overflow -> *
```

It is possible to define separate handlers in each box; uncaught exceptions in a box are handled by a general system handler in an implementation-dependent way (e.g. performing a system-wide reset).

Exceptions should not be raised inside handlers; to ensure this the language definition requires that handlers perform no computation (i.e. the right hand side evaluates to a compile-time constant).

For situations where resources cannot be bounded statically, Hume supports dynamic checks to ensure bounded time and space behaviour: an expression "*expr*

[1] In the implementation this is ensured by re-ordering rules after a successful match.

`within` *time*" or "*expr* `within` *space*" specifies evaluation of *expr* subject to time or space constrains; the exceptions `Timeout`, `HeapOverflow` or `StackOverflow` are raised if the time or space used exceed the specified bounds.

4.1.4 Bounding time and space costs

The network of boxes and wires is fixed at compile time; this means that total time and space resources required for a Hume program can be statically bounded provided that:

1. the resources required by the expressions used in boxes are bounded;
2. the sizes of values in wires are bounded.

The above properties cannot be guaranteed for *arbitrary* programs because the Hume expression layer allows general recursive functions and is therefore Turing-complete. This leaves two alternatives for obtaining bounded time and space behaviour:

Restrict the expression language: prohibit recursive data structures and functions or restrict to guaranteed terminating forms (e.g. primitive recursion); such a language subset would exhibit bounded time and space behaviour by construction at the cost of lower expressiveness;

Employ static analysis: obtain approximate bounds for the use of general recursive functions in *specific* programs, accepting the possibility of failing to derive useful bounds for some programs.

The two approaches are often combined in practice: Hammond and Michaelson (2002) considered a subset designated FSM-Hume resulting from restricting data types to be finite and functions to be first-order and non-recursive; this subset has been shown to correspond to finite state machines (Michaelson et al. 2005). Hammond and Michaelson obtained stack and heap costs bounds for FSM-Hume programs using a source-level analysis that keeps track of maximum costs of expressions. The advantage of a static analysis is that it avoids the combinatorial explosion of associated with enumerating all possible execution traces (which is theoretically possible for FSM-Hume programs).

4.2 A core subset of Hume

Core Hume is a small but representative subset of the Hume language intended as the target for our cost analysis. The motivation for defining this core language is to simplify the rigorous definition of the semantics, analysis and formal proofs. The rationale for defining the core language is to select a kernel of language constructs and to omit redundants ones that can be obtained by translation into the kernel.

A core language can also be seen as an intermediate step in the compilation process, e.g. as in the GHC Haskell compiler (Tolmach 2001); this decoupled approach has several advantages:

- a compiler front-end can translate programs in the full language into the core language; this means that the back-end need only deal with the smaller core language rather than all the syntactical features of a realistic programming language;
- many high-level optimisations can be done in the core language, either during the translation or as a separate program transformation phase;
- compile-time static analysis can be done *after* the core language translation; this not only simplifies the analysis (by restricting it to the smaller core language) but also makes the analysis orthogonal to the front-end; for cost analysis in particular, this ensures that costs incurred by the translation into the core are properly accounted.

The expression layer of the core language has a *strict* and *first order* semantics. Although there have been persuasive arguments in the functional programming community favouring non-strict semantics (Hughes 1989) and an increase in popularity of general-purpose non-strict languages such as Haskell (Jones 2003), we justify the restriction to a strict semantics in order to meet the requirements for tight and predictable resource behaviour.

The restriction to a first order semantics is motived by our size and cost analysis. While uncharacteristic of modern general purpose functional programming languages, this allows us to obtain tight size and space bounds for many non-trivial programs. Thus, we shall argue that the restrictions yield a positive trade-off between expressiveness and predictability for the specific domain of application of real-time embedded systems. Furthermore, we will show by an example in Chapter 7 how the standard *defunctionalisation* program transformation can mitigate this limitation to some extent.

In Chapter 6 we will define a realistic execution model for Core Hume, i.e. an abstract machine that can be directly mapped onto a general-purpose or embedded computer. This execution model will be used to formally specify the stack and heap space cost metrics. Since these are the only dynamic space costs in the abstract machine, bounding stack and heap is enough to guarantee bounded space behaviour for Core Hume programs.

4.2.1 Syntax

Expression notation

The core expression language is a simple first-order functional notation similar to a subset of Standard ML or Haskell and corresponds to a proper subset of the Hume expression language; the abstract syntax is presented in Table 4.1.

The atomic expressions are integers and identifiers; compound expressions are obtained by applications of functions, constructors or primitive operators, local definitions and case expressions.

We use the notation $\vec{e}$ to represent a tuple of arguments $(e_1, \ldots, e_k)$. Note that tuples occur only as arguments of functions, constructors or primitive operations. In particular, the result of a function cannot be a tuple; alternatively, the function can return a product data type. The separation between tuples and data types is motivated by the desire to model space costs of the abstract machine: argument tuples will be allocated in the stack while data types will be allocated in the heap.

Constants other than integers are represented by constructors with no arguments; when it is clear from the context, we omit the empty tuple of arguments, e.g. we write the booleans as True and False instead of $\mathsf{True}\ (\,)$ and $\mathsf{False}\ (\,)$.

Case expressions scrutinise a value against a sequence of pattern-matching alternatives. An alternative $c\ \vec{x} \rightarrow e$ matches the application of constructor c and binds variables $\vec{x}$ in expression e to the constructor arguments. We will assume, without loss of generality, that bound variables $\vec{x}$ are distinct. The tuple of variables in an alternative can be empty (matching a constructed value without arguments); as in applications, we will omit the empty tuple in these alternatives and write $c \rightarrow e$ instead of $c\ () \rightarrow e$.

A case expression decomposes a single constructor application; nested pattern matching must therefore be translated into nested case expressions (Augustsson 1985, Wadler 1987). Case alternatives need not be exhaustive, but each constructor can appear in at most one of the alternatives; therefore, a value can match at most a single alternative and the order among alternatives is not relevant. We will sometimes use a set notation $\{c_i\ \vec{x}_i \rightarrow e_i\}_{i=1}^{n}$ for alternatives. An if-then-else expression

$$\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$$

n	$\in$	**Int**	integers
x, f	$\in$	**Var**	identifiers
p	$\in$	**Prim**	primitive operations
c	$\in$	**Cons**	data constructors
e	$\in$	**Expr**	expressions
e	$::=$	n	integer
	$\mid$	x	identifier
	$\mid$	$f\ \vec{e}$	function application
	$\mid$	$c\ \vec{e}$	constructor application
	$\mid$	$p\ \vec{e}$	primitive application
	$\mid$	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	local definition
	$\mid$	$\mathsf{case}\ e\ \mathsf{of}\ \mathit{alts}$	case expression
$\vec{e}$	$::=$	$(e_1,\ \ldots,\ e_k)$	$(k \geq 0)$
alts	$\in$	**Alts**	alternatives
alts	$::=$	$c_1\ \vec{x}_1 \rightarrow e_1 \mid \ldots \mid c_n\ \vec{x}_n \rightarrow e_n$	$(n \geq 1)$
$\vec{x}$	$::=$	$(x_1, \ldots, x_k)$	$(k \geq 0)$
decl	$\in$	**Decl**	function definitions
decl	$::=$	$\mathsf{let}\ f\ \vec{x} = e$	non-recursive function
	$\mid$	$\mathsf{letrec}\ f\ \vec{x} = e$	recursive function

Table 4.1: Abstract syntax of Core Hume expressions

is used as an abbreviation for a case expression over the boolean data type:

$$\mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{True} \to e_1\ \mid\ \mathsf{False} \to e_2$$

Note also that we disallow case expressions over primitive integers; this makes the syntax and semantics of case alternatives uniform. Case discrimination over integers can be expressed using primitive operations and conditionals, e.g. instead of

$$\mathsf{case}\ e_0\ \mathsf{of}\ n \to e_1\ \mid\ \mathsf{default} \to e_2$$

we write

$$\mathsf{if}\ e_0 = n\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$$

where $=$ is the primitive operation of equality on integers (written in infix form for readability).

In a declaration $\mathsf{let(rec)}\ f\ \vec{x} = e$ the identifier f is bound to a function, $\vec{x}$ are the formal parameters and e is an expression (the function body). To simplify the presentation, but without loss of generality, we syntactically distinguish recursive and non-recursive function definitions. Because the language is first-order, functions are explicitly named and partial applications are disallowed. For simplicity, we consider only single recursively-defined functions[2]; the generalisation to mutually recursive definitions is technically straightforward but cumbersome.

Coordination notation

The syntax of coordination statements is presented in Table 4.2. The coordination language allows the definition of a static network of processes called *boxes*. Each box communicates through a fixed number of *input* and *output* ports by a set of rules defining mappings from inputs to outputs. Rules can be matched *fairly* or *unfairly*; unfair matching is done in top-to-bottom order; fair matching re-orders the rules after a successful match to ensure fairness among patterns.

Each rule is defined by a tuple of input *patterns* and a tuple of output *expressions*. Patterns in box rules can ignore inputs, bind variables and scrutinise a single constructor application. The special asynchronous patterns "*" and "_*" can be used to selectively ignore inputs: "*" ignores an input and does not consume it; "_*" ignores an input but consumes it if it is available. The right-hand side of a box rule is a tuple of expressions; each expression corresponds to the value of one output port. A special symbol "*" represents a "null" value, i.e. the absence of a value for an output port.

[2] Note that in a first-order language mutual recursion cannot be encoded using tuples.

$decl$	$::=$	$\mathsf{box}\ id\ ports\ ports\ match\ rules$	box
	$\mid$	$\mathsf{wire}\ id\ id$	wiring
	$\mid$	$\mathsf{initial}\ id\ e$	initial value
$ports$	$::=$	$(id_1 : \tau_1, \ldots, id_k : \tau_k)$	box ports ($k \geq 1$)
$match$	$::=$	$\mathsf{unfair} \mid \mathsf{fair}$	fair/unfair matching
$rules$	$::=$	$rule_1 \ \mathsf{I}\ \ldots\ \mathsf{I}\ rule_n$	box rules ($n \geq 1$)
$rule$	$::=$	$(ap_1, \ldots, ap_i) \rightarrow (ae_1, \ldots, ae_j)$	box rule ($i \geq 1$, $j \geq 1$)
ae	$::=$	$\mathsf{*} \mid e$	asynchronous expression
ap	$::=$	$\mathsf{*} \mid \_\mathsf{*} \mid x \mid c\ \vec{x}$	asynchronous pattern

Table 4.2: Abstract syntax of Core Hume coordination declarations.

Our core language restricts the Hume definition (Hammond et al. 2007) concerning the terms allowed at the coordination and expression layers:

- the left-hand side of a box rule is not a single pattern but a *tuple* of patterns; e.g. it is therefore not possible to bind a variable to the complete tuple of inputs;
- the right-hand side of a box rule is always a tuple of expressions, *not* an expression that may evaluate to a tuple;
- the null value "*" can occur in a box output tuple but *not* in an expression; in particular, functions cannot evaluate to "*".

These restrictions separate the roles of the coordination and expression layers: the coordination layer checks for availability of inputs and which outputs will be produced; the expression layer is used solely for computation.

The advantage of this separation is a simplification of the language semantics: for example, since "*" is not an expression, the corresponding semantic domain need not include a special "null" point to represent the absence of an output value.[3]

[3] The denotational domain of expression values still needs to include a bottom value to represent runtime errors and non-termination. The null value, however, must be distinct from bottom: the test for null equality is a continuous function while the test for bottom equality is non-continuous.

The clear separation between layers is also beneficial from the point-of-view of the implementation: the absence of the null value allows for more efficient “unboxed” representations for expression values, leaving the boxed representation necessary only at the coordination level.

Boxes are connected into a static process network by *wires*, which are single-buffered communication links connecting box ports together. A single output port can be connected to an arbitrary number of input ports. The wiring is static, meaning that the communication network is totally determined at compile time.

Values in wires can be of arbitrary zero order type, e.g. integers, booleans, tuples, lists or binary trees. For wire data types of variable size (e.g. lists or trees), the compiler or runtime system might require additional information to bound the required wire heap (e.g. user annotations, profiling data or static analysis).

The core language includes a single point-to-point wiring primitive `wire` and a single initialiser primitive `initial`. Each port is identified by an identifier, e.g. *box.port*. The initialiser allows specifying a starting value for a wire, e.g. in a feedback loop.

4.2.2 Type discipline

Core Hume programs must be well-typed according to a first-order fragment of a standard Damas-Milner type system. In this section we describe the type system in some detail, since this will be the basis for the size and cost analyses that will be developed in the following chapters.

Syntax of types

The syntax of types is given by the following grammar.

$$
\begin{array}{lcll}
\tau & \in & \mathbf{Type}_0 & \\
\tau & ::= & \alpha \;\mid\; \mathsf{Int} \;\mid\; \mathsf{D}\,\vec{\tau} & \text{zero order types} \\
\nu & ::= & \tau \;\mid\; \vec{\tau} \rightarrow \tau & \text{first order types} \\
\vec{\tau} & ::= & (\tau_1,\, \ldots,\, \tau_k) & (k \geq 0) \\
\sigma & \in & \mathbf{Type}_1 & \\
\sigma & ::= & \nu \;\mid\; \forall\alpha.\,\sigma & \text{quantified types} \\
\alpha & \in & \mathbf{TVar} & \\
\alpha & ::= & \mathsf{a} \;\mid\; \mathsf{b} \;\mid\; \mathsf{c} \;\mid\; \ldots & \text{type variables}
\end{array}
$$

Type variables are taken from a countable set **TVar**; we use lowercase letters from

$$\begin{aligned}
\mathrm{FTV}(\alpha) &= \{\alpha\} \\
\mathrm{FTV}(\mathsf{Int}) &= \emptyset \\
\mathrm{FTV}(\mathsf{D}\,\vec{\tau}) &= \mathrm{FTV}(\vec{\tau}) \\
\mathrm{FTV}((\tau_1,\,\ldots,\,\tau_k)) &= \textstyle\bigcup_{i=1}^{k} \mathrm{FTV}(\tau_i) \\
\mathrm{FTV}(\vec{\tau} \to \tau) &= \mathrm{FTV}(\vec{\tau}) \cup \mathrm{FTV}(\tau) \\
\mathrm{FTV}(\forall\alpha.\,\sigma) &= \mathrm{FTV}(\sigma) \setminus \{\alpha\}
\end{aligned}$$

Table 4.3: Free type variables

the beginning of the Greek alphabet α, β as syntactical meta-symbols for type variables.

Zero order types are either variables, the primitive type of integers, or an application of a data type constructor to a sequence of types; data type constructors are described in Section 4.2.2.

First order types $(\tau_1,\,\ldots,\,\tau_k) \to \tau_{k+1}$ is the type of a function with k arguments of types $\tau_1,\,\ldots,\,\tau_k$ and result of type τ_{k+1}. Note that argument tuples can be empty, e.g. for data constructors with zero arguments.

Types can be quantified over type variables, yielding *type schemes.* We use the universal logic quantifier $\forall$ for type quantification and abbreviate the notation, writing $\forall\vec{\alpha}.\,\sigma \overset{\text{def}}{=} \forall\alpha_1.\,\ldots\forall\alpha_n.\,\sigma$ to quantify over a sequence $\vec{\alpha} = (\alpha_1,\,\ldots,\,\alpha_n)$ of type variables.

The set of type variables with free occurrences in a type scheme σ is $\mathrm{FTV}(\sigma)$. Table 4.3 defines $\mathrm{FTV}(\sigma)$ inductively on the structure of σ. We will sometimes consider that $\mathrm{FTV}(\sigma)$ yields a sequence of type variables $\vec{\alpha}$ rather than a set, e.g. by ordering the variables in lexicographic order.

Data type declarations

Algebraic types are defined by *data type declarations* with the syntax:

$$\mathit{datadecl} \;::=\; \mathsf{data}\ \mathsf{D}\,\vec{\alpha} = c_1\,\vec{\tau}_1 \;\mid\; \cdots \;\mid\; c_n\,\vec{\tau}_n$$

The declaration defines a type constructor D of arity $|\vec{\alpha}|$ as a sum of n alternatives, each labelled by a unique constructor c_i. The declaration defines a recursive type if the type D occurs in the right-hand side alternatives. Since the expression language is strict, data type definitions cannot be co-inductive: all data type definitions should be well-founded, i.e. have at least one non-recursive alternative.

Example 4.1 The type of booleans and homogeneous lists are declared as

$$\mathsf{data\ Bool}\ () = \mathsf{True}\ ()\ \mid\ \mathsf{False}\ ()$$

$$\mathsf{data\ List}\ (a) = \mathsf{Nil}\ ()\ \mid\ \mathsf{Cons}\ (a,\ \mathsf{List}\ a)$$

which defines True and False as boolean values, the empty list constructor Nil and the list Cons (h, t) with first element h and tail t. □

Type assumptions

As usual in type assignment disciplines, we use *type assumptions* to associate types to free variables. Type assumptions Γ are sequences of pairs $x : \sigma$ or $c : \sigma$ associating type scheme σ with, respectively, an identifier x or constructor c:

$$\Gamma\ ::=\ []\ \mid\ x : \sigma, \Gamma\ \mid\ c : \sigma, \Gamma$$

We will represent the concatenation of two sequences Γ' and Γ by $(\Gamma',\ \Gamma)$; the definition of concatenation is standard and we therefore omit it.

An assumption sequence defines a partial finite map from names to type schemes. We will use the notation $\Gamma(id)$ for the first assumption (if any) associated with id in Γ:

$$(id' : \sigma,\ \Gamma)(id) \overset{\text{def}}{=} \begin{cases} \sigma & \text{if } id = id' \\ \Gamma(id) & \text{otherwise.} \end{cases}$$

Note that $\Gamma(id)$ is undefined when Γ is the empty sequence.

Type substitutions

A *type substitution* is a function $\theta : \mathbf{TVar} \to \mathbf{Type}_0$ from type variables to zero-order types. The *domain* of θ is $\mathrm{dom}(\theta) = \{\alpha \in \mathbf{TVar} : \theta\alpha \neq \alpha\}$, i.e. the set of variables where θ is not the identity. We use the notation $[\alpha \mapsto \tau]$ for the substitution mapping α to τ and behaving as the identity everywhere else, i.e.

$$[\alpha \mapsto \tau]\,\alpha' = \begin{cases} \tau\,, & \text{if } \alpha = \alpha' \\ \alpha' & \text{otherwise .} \end{cases}$$

Substitutions extend from variables to types and schemes in the usual way.

Typing rules and derivations

Tables 4.4 and 4.5 present the typing rules for Core Hume expressions and declarations. Each rule has the form $\{A_1, A_2, \ldots, A_n\}/B$ where $\{A_1, A_2, \ldots, A_n\}$ is a finite set of *antecedents* and B is the *consequent*. We write each rule using the style of natural deduction: the antecedents and consequent are separated by an horizontal line and the rule is labelled with a unique name for easier reference in proofs.

$$[\mathit{label}] \quad \frac{A_1 \quad A_2 \ \cdots \ A_n}{B}$$

A rule $\emptyset/B$ with an empty set of premises is an *axiom* and is written as a single consequent without the horizontal line. Rules are usually specified with meta-variables that can be replaced by terms of the appropriate syntactical kind. Some rules are subject to side conditions to restrict allowed instances.

Given a set of rules, a *derivation* is a finite tree where each internal node is an instance of a rule and each leaf is an instance of an axiom.

We will often prove properties using *induction on derivations* (Winskel 1993, Mitchell 1996). To prove that property P holds for all consequences of derivations we need only show that: (1) P holds for all axioms; and (2) for each rule $\{A_1, \ldots, A_n\}/B$, $P(B)$ holds under the assumptions $P(A_1), \ldots, P(A_n)$.

Description of the typing rules

Types are assigned to expressions by a judgement $\Gamma \vdash_{\mathsf{DM}} e : \tau$ which informally states that under assumptions Γ, e admits type τ. Most of the rules of Table 4.4 are directly adapted from the literature (Milner 1976, Damas and Milner 1982, Damas 1985, Pierce 2002) and we therefore describe them only briefly:

- Rule [*Var*] infers a type for an identifier if it can be obtained from the assumption set Γ using an auxiliary judgement $\Gamma \vdash_{\mathsf{INST}} id : \sigma$ for type scheme instantiation.[4]

- Rules [*FunAp*] and [*ConsAp*] infers a type for an function and constructor applications if the type of domain and argument match.

- Rule [*Let*] infer a type an expression under a local definition. Because the language is first-order, the bound value is zero-order and therefore the rule restricts its type to be monomorphic. Polymorphic definitions are allowed only in function declarations.

[4] A separate judgement is needed because expression types are zero-order while type schemes are first-order.

$$\boxed{\Gamma \vdash_{\mathsf{DM}} e : \tau}$$

$$[\mathit{Int}] \quad \Gamma \vdash_{\mathsf{DM}} n : \mathsf{Int}$$

$$[\mathit{Var}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} x : \tau}{\Gamma \vdash_{\mathsf{DM}} x : \tau}$$

$$[\mathit{FunAp}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} f : \vec{\tau} \to \tau' \qquad \Gamma \vdash_{\mathsf{DM}} \vec{e} : \vec{\tau}}{\Gamma \vdash_{\mathsf{DM}} f\ \vec{e} : \tau'}$$

$$[\mathit{ConsAp}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} c : \vec{\tau} \to \tau' \qquad \Gamma \vdash_{\mathsf{DM}} \vec{e} : \vec{\tau}}{\Gamma \vdash_{\mathsf{DM}} c\ \vec{e} : \tau'}$$

$$[\mathit{Let}] \quad \frac{\Gamma \vdash_{\mathsf{DM}} e_1 : \tau_1 \qquad x : \tau_1, \Gamma \vdash_{\mathsf{DM}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{DM}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$[\mathit{Case}] \quad \frac{\begin{array}{l}\Gamma \vdash_{\mathsf{DM}} e_0 : \tau' \\ \Gamma \vdash_{\mathsf{INST}} c_i : \vec{\tau}_i'' \to \tau' \qquad \vec{x}_i : \vec{\tau}_i'', \Gamma \vdash_{\mathsf{DM}} e_i : \tau \qquad (\forall i)\end{array}}{\Gamma \vdash_{\mathsf{DM}} \mathsf{case}\ e_0\ \mathsf{of}\ \{c_i\ \vec{x}_i \to e_i\}_{i=1}^{n} : \tau}$$

$$\boxed{\Gamma \vdash_{\mathsf{DM}} \vec{e} : \vec{\tau}}$$

$$[\mathit{Tuple}] \quad \frac{\Gamma \vdash_{\mathsf{DM}} e_i : \tau_i \qquad (\forall i)}{\Gamma \vdash_{\mathsf{DM}} (e_1, \ldots, e_k) : (\tau_1, \ldots, \tau_k)}$$

$$\boxed{\Gamma \vdash_{\mathsf{INST}} id : \sigma} \qquad \text{Assumption instantiation}$$

$$[\mathit{Axiom}] \quad \Gamma \vdash_{\mathsf{INST}} id : \Gamma(id)$$

$$[\mathit{Elim}\forall] \quad \frac{\Gamma \vdash_{\mathsf{INST}} id : \forall \alpha.\, \sigma}{\Gamma \vdash_{\mathsf{INST}} id : [\alpha \mapsto \tau]\, \sigma}$$

Table 4.4: Typing rules for Core Hume expressions.

$$\boxed{\Gamma \vdash_{\mathsf{DM}} \mathit{decl} : \sigma}$$

$$[\mathit{Fun}] \quad \frac{x_1 : \tau_1,\, \ldots,\, x_k : \tau_k,\, \Gamma \vdash_{\mathsf{DM}} e : \tau_{k+1}}{\Gamma \vdash_{\mathsf{DM}} \mathsf{let}\ f\ \ (x_1, \ldots x_k) = e\ :\ \forall \vec{\alpha}.\, (\tau_1, \ldots, \tau_k) \to \tau_{k+1}}$$

$$[\mathit{Rec}] \quad \frac{x_1 : \tau_1,\, \ldots,\, x_k : \tau_k,\, f : (\tau_1, \ldots, \tau_k) \to \tau_{k+1},\, \Gamma \vdash_{\mathsf{DM}} e : \tau_{k+1}}{\Gamma \vdash_{\mathsf{DM}} \mathsf{letrec}\ f\ \ (x_1, \ldots, x_k) = e\ :\ \forall \vec{\alpha}.\, (\tau_1, \ldots, \tau_k) \to \tau_{k+1}}$$

$$\text{where } \vec{\alpha} = \left(\bigcup_{i=1}^{k+1} \mathrm{FTV}(\tau_i)\right) \setminus \mathrm{FTV}(\Gamma)$$

Table 4.5: Typing rules for Core Hume function declarations.

- Rule [*Case*] infers a type for a case expression: the type for the scrutinised expression e_0 and co-domain of the constructors c_i must match; each alternative is then typed under extended assumptions for the patterns variables;[5] finally, the types of the right-hand side expressions e_i must be equal among all the alternatives.

The typing judgement $\Gamma \vdash_{\mathsf{DM}} \mathit{decl} : \sigma$ derives a type scheme σ for a function declaration *decl* under assumptions Γ.

A function declaration $\mathsf{let(rec)}\ f\ \ (x_1, \ldots, x_k) = e$ binds the identifier f to a k-argument function with formal parameters $x_1, \ldots, x_k$ and body e. The inferred type has the form $(\tau_1, \ldots, \tau_k) \to \tau_{k+1}$, where $\tau_1, \ldots, \tau_k$ are the types of the arguments and τ_{k+1} is the type of result. The rule [*Rec*] for recursive functions differs from the non-recursive one [*Fun*] in the assumption for f in the typing context of e.

As usual in type systems based on Damas-Milner let-bound polymorphism, the types of functions are quantified in all variables that do not occur free in the type assumptions.

4.2.3 Semantics

In order to formulate and prove the correctness of our analysis, we will present a formal semantics for Core Hume. In this section we focus on the semantics of the expression layer in the form of a standard call-by-value denotational semantics (see Appendix A for a review of the required order-theoric concepts). The semantics for the coordination layer will be presented in Chapter 6.

[5] When $|\vec{x}| = |\vec{\tau}|$ we write $\vec{x} : \vec{\tau}$ as an abbreviation for the assumptions $x_1 : \tau_1, \ldots, x_n : \tau_n$, [].

Semantic domains

Values of Core Hume expressions are integers, constructors and sequences of values. Formally, we define the set of values inductively by a grammar.

Definition 4.2 *The set of* expressible values $\mathbf{V}$ *is defined by the grammar*

$$\begin{array}{rcl} v & \in & \mathbf{V} \\ v & ::= & n \mid c \mid \mathbf{u} \mid \langle v,\, v \rangle \\ c & \in & \mathbf{Cons} \\ n & \in & \mathbb{Z} \end{array}$$

where $\mathbb{Z}$ *is the set of integers and* **Cons** *is the set of data constructors and* $\mathbf{u}$ *is a distinguished unit value.*

We introduce a shorthand notation for multi-arity tuples: $\langle v_1, \ldots, v_n \rangle$ is an abbreviation for $\langle v_1, \langle \ldots, \langle v_{n-1}, v_n \rangle \rangle \rangle$ for $n > 0$. In particular, a singleton $\langle v \rangle$ is just the value v.

The set of denotable values is $\mathbf{V}_\perp = \{\lfloor v \rfloor : v \in \mathbf{V}\} \cup \{\perp\}$ containing the lifted elements of $\mathbf{V}$ together with a distinguished element $\perp$ representing partiality (i.e. run-time errors and non-termination).

Definition 4.3 *The domain of* denotations of expressions *is the flat CPO* $(\mathbf{V}_\perp, \sqsubseteq)$*, where* $u \sqsubseteq v \iff u = \perp \vee u = v$*.*

The denotations of functions are continuous mappings from $\mathbf{V}$ to $\mathbf{V}_\perp$, or equivalently elements of $[\mathbf{V} \to \mathbf{V}_\perp]$. Note that $\perp$ is not included in the domain of function denotations: since we are modelling a strict semantics the image of $\perp$ is always $\perp$. The set $[\mathbf{V} \to \mathbf{V}_\perp]$ with the pointwise order $\sqsubseteq$ is a CPO (see Section A.1.7).

Environments

To associate denotations to identifiers we employ the standard notion of an *environment.* Since our language semantics is first-order, we distinguish environments ρ for basic values from environments φ for functional values:

$$\rho \in \mathbf{Env} \overset{\text{def}}{=} \mathbf{Var} \to \mathbf{V}_\perp \tag{4.1}$$

$$\varphi \in \mathbf{Fenv} \overset{\text{def}}{=} \mathbf{Var} \to [\mathbf{V} \to \mathbf{V}_\perp] \tag{4.2}$$

We denote by ρ_0 the environment that maps every identifier to bottom (i.e. $\rho_0(x) = \perp$ for all $x \in \mathbf{Var}$), and use the notations $\rho[x \mapsto v]$ and $\varphi[f \mapsto F]$, where

$v \in \mathbf{V}$ and $F \in [\mathbf{V} \to \mathbf{V}_\perp]$, for extending environments as follows:

$$\rho[x \mapsto v](x') = \begin{cases} \lfloor v \rfloor & \text{if } x = x' \\ \rho(x') & \text{otherwise} \end{cases} \qquad \varphi[f \mapsto F](f') = \begin{cases} F & \text{if } f = f' \\ \varphi(f') & \text{otherwise} \end{cases}$$

Note that, because the Core Hume semantics is strict, we will never extend an environment ρ with a bottom value.

Semantic equations

The semantics of Core Hume is given in Table 4.6 by functions $\mathcal{E}$ for expressions, $\mathcal{A}$ for alternatives, $\mathcal{M}$ for pattern matching and $\mathcal{D}$ for declarations. We use a metalanguage based on the λ-calculus to define the semantics functions (Stoy 1977, Winskel 1993):

Lambda-abstraction: If $E \in \mathbf{V}_\perp$ is a continuous expression in a variable $x \in \mathbf{V}$ then $\lambda x.\, E$ denotes a function in $[\mathbf{V} \to \mathbf{V}_\perp]$.

Function lifting: Let $F \in [\mathbf{V} \to \mathbf{V}_\perp]$ and define the *lifting* of F to be F^*:

$$F^*(v) = \begin{cases} F(v') & \text{if } v = \lfloor v' \rfloor \\ \perp & \text{if } v = \perp \end{cases}$$

Then $F^* \in [\mathbf{V}_\perp \to \mathbf{V}_\perp]$.

Strict binding: If $E_1 \in \mathbf{V}_\perp$ and $E_2 \in \mathbf{V}_\perp$ are continuous expressions in all variables, then the expression

$$\text{let } x \Leftarrow E_1.\ E_2 \stackrel{\text{def}}{=} (\lambda x.\, E_2)^*\, E_1$$

is also continuous in all variables (since it is defined by λ-abstraction, lifting and application).

Case expressions: If $e \in \mathbf{V}$ and $E_1, \ldots, E_n \in \mathbf{V}_\perp$ are continuous expressions and $c_1, \ldots, c_n$ are all distinct, then

$$\text{case } e \text{ of } \{c_1.\, E_1 \mid \cdots \mid c_n.\, E_n\} = \begin{cases} E_i & \text{if } v = c_i \text{ for some } i \\ \perp & \text{otherwise} \end{cases}$$

is a continuous expression in all variables.

Tuple binding: We use also a form of lambda-abstraction that binds components of tuples. If E is in $\mathbf{V}_\perp$ and is continuous in $x_1, \ldots, x_k$, then the lambda-expression $\lambda\langle x_1, \ldots, x_k\rangle.\, E$ represents the function

$$(\lambda\langle x_1, \ldots, x_k\rangle.\, E)\ v = \begin{cases} E[x_1 \mapsto v_1, \ldots\ x_k \mapsto v_k] & \text{if } v = \langle v_1, \ldots, v_k\rangle \\ \perp & \text{otherwise} \end{cases}$$

It is immediate that this function is continuous and, therefore, in $[\mathbf{V} \to \mathbf{V}_\perp]$.

$$
\begin{aligned}
\mathbf{Env} &\stackrel{\text{def}}{=} \mathbf{Var} \to \mathbf{V}_{\perp} \\
\mathbf{Fenv} &\stackrel{\text{def}}{=} \mathbf{Var} \to [\mathbf{V} \to \mathbf{V}_{\perp}]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E} &: \mathbf{Expr} \to \mathbf{Fenv} \to \mathbf{Env} \to \mathbf{V}_{\perp} \\
\mathcal{E}\llbracket\, n \,\rrbracket\ \varphi\ \rho &\stackrel{\text{def}}{=} \lfloor n \rfloor \\
\mathcal{E}\llbracket\, x \,\rrbracket\ \varphi\ \rho &\stackrel{\text{def}}{=} \rho(x) \\
\mathcal{E}\llbracket\, c\ \vec{e} \,\rrbracket\ \varphi\ \rho &\stackrel{\text{def}}{=} \text{let } v \Leftarrow \mathcal{E}\llbracket\, \vec{e} \,\rrbracket\ \varphi\ \rho.\ \lfloor \langle c,\, v\rangle \rfloor \\
\mathcal{E}\llbracket\, f\ \vec{e} \,\rrbracket\ \varphi\ \rho &\stackrel{\text{def}}{=} \text{let } v \Leftarrow \mathcal{E}\llbracket\, \vec{e} \,\rrbracket\ \varphi\ \rho.\ \varphi_f\ v \\
\mathcal{E}\llbracket\, \textsf{let}\ x = e_1\ \textsf{in}\ e_2 \,\rrbracket\ \varphi\ \rho &\stackrel{\text{def}}{=} \text{let } v \Leftarrow \mathcal{E}\llbracket\, e_1 \,\rrbracket\ \varphi\ \rho.\ \mathcal{E}\llbracket\, e_2 \,\rrbracket\ \varphi\ \rho[x \mapsto v] \\
\mathcal{E}\llbracket\, \textsf{case}\ e\ \textsf{of}\ \mathit{alts} \,\rrbracket\ \varphi\ \rho &\stackrel{\text{def}}{=} \text{let } v \Leftarrow \mathcal{E}\llbracket\, e \,\rrbracket\ \varphi\ \rho.\ \mathcal{A}\llbracket\, \mathit{alts} \,\rrbracket\ \varphi\ \rho\ v
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}\llbracket\, () \,\rrbracket\ \varphi\ \rho &\stackrel{\text{def}}{=} \lfloor \mathbf{u} \rfloor \\
\mathcal{E}\llbracket\, (e_1, \ldots, e_k) \,\rrbracket\ \varphi\ \rho &\stackrel{\text{def}}{=} \ \text{let } v_1 \Leftarrow \mathcal{E}\llbracket\, e_1 \,\rrbracket\ \varphi\ \rho. \\
&\qquad\ \ \vdots \\
&\qquad \text{let } v_k \Leftarrow \mathcal{E}\llbracket\, e_k \,\rrbracket\ \varphi\ \rho.\ \lfloor \langle v_1, \ldots, v_k \rangle \rfloor
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A} &: \mathbf{Alts} \to \mathbf{Fenv} \to \mathbf{Env} \to \mathbf{V} \to \mathbf{V}_{\perp} \\
\mathcal{A}\llbracket\, \{c_i\ \vec{x}_i \to e_i\}_{i=1}^{k} \,\rrbracket\ \varphi\ \rho &\stackrel{\text{def}}{=} \ \lambda\langle v',\, v\rangle.\ \text{case } v' \text{ of} \\
&\qquad\quad c_1.\ \mathcal{M}\llbracket\, c_1\ \vec{x}_1 \to e_1 \,\rrbracket\ \varphi\ \rho\ v \quad | \\
&\qquad\qquad \vdots \\
&\qquad\quad c_k.\ \mathcal{M}\llbracket\, c_k\ \vec{x}_k \to e_k \,\rrbracket\ \varphi\ \rho\ v \\
\mathcal{M} &: \mathbf{Alt} \to \mathbf{Fenv} \to \mathbf{Env} \to \mathbf{V} \to \mathbf{V}_{\perp} \\
\mathcal{M}\llbracket\, c\ (x_1, \ldots, x_k) \to e \,\rrbracket\ \varphi\ \rho &\stackrel{\text{def}}{=} \lambda\langle v_1, \ldots, v_k\rangle.\ \mathcal{E}\llbracket\, e \,\rrbracket\ \varphi\ \rho[x_1 \mapsto v_1,\ \ldots,\ x_k \mapsto v_k]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{D} &: \mathbf{Decl} \to \mathbf{Fenv} \to [\mathbf{V} \to \mathbf{V}_{\perp}] \\
\mathcal{D}\llbracket\, \textsf{let}\ f\ (x_1, \ldots, x_k) = e \,\rrbracket\ \varphi &\stackrel{\text{def}}{=} \lambda\langle v_1, \ldots, v_k\rangle.\ \mathcal{E}\llbracket\, e \,\rrbracket\ \varphi\ \rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] \\
\mathcal{D}\llbracket\, \textsf{letrec}\ f\ (x_1, \ldots, x_k) = e \,\rrbracket\ \varphi &\stackrel{\text{def}}{=} \mathit{fix}(\mathcal{F}), \\
\text{where } \mathcal{F} = \lambda F.\ \lambda\langle v_1, \ldots, v_k\rangle.&\ \mathcal{E}\llbracket\, e \,\rrbracket\ \varphi[f \mapsto F]\ \rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]
\end{aligned}
$$

Table 4.6: Denotational semantics for Core Hume expressions and functions.

Least fixed point: If D is a pointed CPO and $F \in [D \to D]$ (i.e. F is a continuous function in D), then $\mathit{fix}(F)$ is the least fixed point of F in D.

Because Core Hume is first-order, functions definitions are only allowed at the outermost level of declarations. The semantics of non-recursive functions is defined by a lambda-abstraction denoting an element of $[\mathbf{V} \to \mathbf{V}_\perp]$. The semantics of recursive functions is defined as the least fixed point of a functional in $[[\mathbf{V} \to \mathbf{V}_\perp] \to [\mathbf{V} \to \mathbf{V}_\perp]]$. The following lemma establishes the continuity of the functional, which guarantees the existence of the least fixed point (Davey and Priestley 1990).

Lemma 4.4 *$\mathcal{E}$ is continuous in its φ, ρ arguments.*

Proof: Analogous to standard proofs in semantics textbooks such as (Stoy 1977, Winskel 1993). □

4.3 Concrete syntax

A Core Hume program consists of a sequence of data type declarations, function declarations and coordination declarations. In this section we describe the concrete syntax of Core Hume programs; this is mostly a subset of the syntax of Hume (Hammond et al. 2007) with some minor exceptions to simplify parsing (e.g. the requirement that data type constructors start with upper-case letters).

4.3.1 Lexical conventions

Identifiers start with a lowercase letter followed by any sequence of letters, digits, underscore (`_`) and quote (`'`) characters.

Constructors start with uppercase letters, followed by any sequence of letters, digits and underscore. The exception to the latter rule are the list constructors, where we use a Haskell-like notation: `[]` is the empty list and h`:`t is the list with head h and tail t.

Types type variables start with a lowercase letters; data type constructors start with uppercase letters. The list type is written `[`τ`]`. An n-argument function type is written `{`τ_1`,` τ_2`,` `...``,` $\tau_n$`}->`$\tau_{n+1}$; a single argument function can be written $\tau_1$`->`τ_2 omitting the braces around the argument type.

Comments single-line comments start with `--` and extend to the end-of-line. Block comments are enclosed within `{-` and `-}`.

4.3.2 Syntactical conventions

Application of functions and constructors is written by juxtaposition, i.e. $f\ e_1\ \ldots\ e_k$ instead of $f\ (e_1, \ldots, e_k)$.

Parenthesis `(` and `)` are used to delimit expressions when required.

Integer arithmetic operators `+`, `-`, `*`, `==`, `<=` are written in infix notation and follow the usual precedences. Function and constructor applications have higher precedence than infix operators, thus `foo x+y` parses like `(foo x)+y`.

Floating point arithmetic operators are written `+.`, `*.`, `/.`, etc. to lexically distinguish them from the operators on integers.

List extensions is a shorthand notation for nested application of the list constructors: `[`e_1, e_2, …, e_n`]` stands for e_1`:`e_2`:` … e_n`:[]`.

Function definitions can be made using equations with patterns on the left-hand sides; each equation is terminated by a semicolon. Pattern matching equations are translated into case expressions using the algorithm of Wadler (1987).

Type signatures are required for each function; this is done solely to to simplify the size and cost analysis presented in the the following chapters. In any case, type signatures could be introduced automatically, if desired, using the Damas-Milner type reconstruction algorithm (Damas 1985).

4.4 Examples

We shall now present some example programs in Core Hume, mainly to familiarise the reader with the concrete syntax used. Some of the examples (e.g. the list reverse function) will also be revisited in Chapter 7 when we use them as subjects for the size and cost analyses of Chapters 5 and 6.

4.4.1 List functions

Our first examples are standard list functions from the Haskell prelude: the `head` and `tail` projections and list reverse written in tail-recursive style using an accumulating parameter.

```
-- head and tail are not defined for []
head :: [a] -> a
head (x:xs) = x;
```

```
tail :: [a] -> [a]
tail (x:xs) = xs;

-- reverse using an accumulating parameter
reverse :: [a] -> [a]
reverse xs = revAcc xs [];

revAcc :: {[a],[a]} -> [a]
revAcc [] ys = ys;
revAcc (x:xs) ys = revAcc xs (x:ys);
```

Pattern matching on the left-hand side of these equations is translated in the Core language using the algorithm of Wadler (1987). For example, the compilation of *revAcc* yields the following abstract syntax:

$$\begin{aligned} \mathsf{letrec}\ \mathit{revAcc}\ (xs, ys) \ &= \mathsf{case}\ xs\ \mathsf{of} \\ &\quad \mathsf{Nil} \rightarrow ys \\ &\mid \mathsf{Cons}\ (x, xs') \rightarrow \mathit{revAcc}\ (xs', \mathsf{Cons}\ (x, ys)) \end{aligned}$$

4.4.2 Parity checker

Our next example is a Core Hume box implementing a *parity checker*: a box processing a stream of bits one bit at a time and keeping a record of the parity of the number of True bits. This is a very simple example of a reactive process using a wire with a feedback to record state (the current parity).

```
not :: Bool -> Bool
not True = False ;
not False = True ;

box parity
in (bit::Bool, state::Bool) -- inputs
out (state'::Bool)          -- output
match
   (True, p) -> (not p) -- invert parity
   (False,p) -> (p) ;   -- maintain parity

wire parity.state parity.state' ; -- feedback loop
initial parity.state False ;      -- initial parity
```

```
-- traffic light states
data Light = Red | Amber | Green | Red_Amber ;

-- traffic lights delay
data Delay = Done | Wait Int ;

-- delay values (in clock cycles, e.g. seconds)
red_amber_delay = 2 ;
green_delay     = 90 ;
amber_delay     = 2 ;
red_delay       = 90 ;

countdown :: Int -> Delay
countdown ticks = if ticks>0 then Wait (ticks-1) else Done ;

-- traffic lights controller
box lights
in (clk::(), delay::Int, state::Light)
out (delay’::Int, state’::Light)
match
   (_, Done, Red) ->       (Wait red_amber_delay, Red_Amber)
|  (_, Done, Red_Amber) -> (Wait green_delay, Green)
|  (_, Done, Green) ->     (Wait amber_delay, Amber)
|  (_, Done, Amber) ->     (Wait red_delay, Red)
|  (_, Wait ticks, state) -> (countdown ticks, state) ;

wire lights.delay lights.delay’;
wire lights.state lights.state’;
-- clock input wired to light.clk

initial lights.state Red;
initial lights.delay (Wait red_delay);
```

Figure 4.1: Traffic lights controller in Core Hume.

4.4.3 Traffic lights controller

The next example in Figure 4.1 is a finite-state controller for a set of traffic lights. The controller goes through a sequence of states: red (stop), red and amber (prepare to go), green (go) and amber (prepare to stop). The state transitions are delayed by multiples of a clock input. The clock is a pure synchronisation signal; therefore its value (of type unit) is not used.

Summary

We have introduced Core Hume, a small but representative subset of the Hume language. Core Hume programs consist of a static network of processes described in a strict, first-order, statically-typed applicative language.

We have presented formal definitions of the syntax, semantics and type system of Core Hume. These definitions form the basis for the specification and correctness proofs of the size and costs analyses developed in Chapters 5 and 6.

Chapter 5

Size analysis

In this chapter we present an size analysis for Core Hume programs based on a sized type system. This purely denotational size analysis is a preliminary step towards the cost analysis for recursive programs which will be developed in Chapter 6.

We present the syntax and semantics of sized types (Section 5.2), the type inference rules (Section 5.3) and prove that the analysis is sound with respect to the denotational semantics defined in Chapter 4 (Section 5.4). We also present an algorithm for reconstructing sized types automatically (Section 5.5) and dicuss limitations of the analysis regarding quality of the size information obtained (Section 5.6).

5.1 Overview

Size analyses are static analysis techniques for automatically obtaining predictive information about the sizes of data values in a program. The basic idea is to attach *size measures* to data structures and express the effect of a program as a size relation.

For example, consider two Core Hume recursive functions for list concatenation and insertion (written in a Haskell-style for legibility):

```
append :: {[a],[a]} -> [a]
append [] ys = ys ;
append (x:xs) ys = x:append xs ys;

insert :: {Int,[Int]} -> [Int]
insert x [] = [x] ;
insert x (y:ys) = if x<y then x:y:ys
                  else if x>y then y:insert x ys
```

```
            else y:ys;
```

Defining the size of a list to be the *length*, i.e.

$$\begin{aligned} |\texttt{[]}| &= 0 \\ |x : xs| &= 1 + |xs|\,, \end{aligned}$$

we can verify (e.g. by structural induction) that the following size relations hold:

$$|\texttt{append}\ xs\ ys| = |xs| + |ys| \tag{5.1}$$

$$|ys| \leq |\texttt{insert}\ x\ ys| \leq 1 + |ys| \tag{5.2}$$

Equation (5.1) gives the size of the result list as the sum of two input lists. Inequations (5.2) give lower-and upper-bounds on the size of the result list (the exact size will depend on whether the inserted value occurs in the input list).

In the sized type system of Chin and Khoo (2001), such relations are expressed by the following annotated types:[1]

$$\mathit{append} : \langle\{[a]^i, [a]^j\} \to [a]^k,\ i + j = k\rangle \tag{5.3}$$

$$\mathit{insert} : \langle\{\mathsf{Int}, [\mathsf{Int}]^n\} \to [\mathsf{Int}]^m,\ n \leq m \wedge m \leq n + 1\rangle \tag{5.4}$$

These examples illustrate three characteristics of size analysis:

Relational information: sizes of individual data structures are represented by *annotations* in types; the size analysis express not just sizes of individual values, but also *relations* between sizes of values;

Arithmetic relations: size relations are naturally expressed using arithmetic constraints; to allow automatic manipulation these are typically restricted to decidable fragments, e.g. Presburger arithmetic (Pugh 1992);

Approximate information: the size analysis must allow approximations; for example, the size m of the result of insert depends on which branch of the dynamic test are taken; the analysis obtains a range as a conjunction of two inequations $n \leq m \wedge m \leq n + 1$.

Our sized analysis is based on the sized type systems presented by Hughes, Pareto and Sabry (1996) and Chin and Khoo (2001); however, it differs from these works in several aspects:

1. we do *inference* of size information rather than just *checking* programmer annotations as in the system of Hughes et al.;

[1] We ignore the issues of size and type polymorphism in this example; these will be addressed in the technical development of Section 5.2.

2. our sized type system infers a *safety* property i.e. safe bounds on sizes of dynamic values; it does *not* prove termination as does the system of Hughes et al.;

3. like the system of Chin and Khoo (2001), we obtain size relations for recursive function automatically; however, we do so using standard fixpoint approximation techniques (Cousot and Halbwachs 1978) rather than the transitive closure used by Chin and Khoo;

4. we generalise the notions of size in both previous approaches and allow user defined metrics for new data types; this extends the applicability of size analysis to non-linear data structures (e.g. binary trees);

5. our soundness proof for the type system corrects the one by Chin and Khoo which erroneously assumes completeness of the lattice of constraints.

The presentation proceeds as follows: in Section 5.2 we introduce the syntax of annotated types and size constraints; in Section 5.3 we present the typing rules for deriving sized types for Core Hume programs; in Section 5.4 we present a size semantics based on the denotational semantics of Core Hume and prove the correctness of the type system; in Section 5.5 we present the sized type inference algorithm. Finally, Section 5.6 discusses some limitations of the size analysis.

5.2 Size analysis for Core Hume

5.2.1 Sizes of data types

To extend Core Hume with sized types we need to choose size measures. As in Dependent ML (Xi 1998), we rely on the programmer specifying a size measure for user-defined data types by augmenting the type declarations with size variables and constraints. In the absence of a user-defined size, the data type is treated as unsized. This allows the programmer to assign meaningful size measures to the relevant data types and avoid size information when it is not required.

For example, the list data type of Example 4.1 can be extended with a measure for *list length* by the declaration:

$$\begin{array}{lcll} \mathsf{data}\ \mathsf{List}^i\, a & = & \mathsf{Nil} & \{i = 0\} \\ & \mid & \mathsf{Cons}\ (a, \mathsf{List}^j\, a) & \{i = 1 + j \wedge 0 \leq j\} \end{array}$$

The variables i, j represent sizes of lists; the Nil and Cons constructors have been augmented with constraints on these variables (the syntax of constraints will be formalised in the next section). The constraint $i = 0$ in the first branch encodes the relation $|\mathsf{Nil}| = 0$; the constraint $i = 1 + j$ in the second branch encodes the relation $|\mathsf{Cons}(x, xs)| = 1 + |xs|$; the constraint $0 \leq j$ encodes the non-negativity of list lengths.

Note that we can also define sizes for simple enumerations, e.g. by associating each constructor with an integer size; this is used on case selection to propagate information. For example, we can define the size of booleans as $|\mathsf{True}| = 1$ and $|\mathsf{False}| = 0$:

$$\mathsf{data\ Bool}^i = \mathsf{True}\ \{i = 1\} \quad | \quad \mathsf{False}\ \{i = 0\}$$

The only type with a predefined size measure is Int, the primitive type of integers: we take $|n| = n$, i.e. the size of an integer is its own value.

The reader may wonder why we did we not choose a logarithmic size for integers (e.g. $|n| = \lceil \log_2(n+1) \rceil$); the rationale for our choice is that the size of integers is not intended to bound storage (we will instead assume integers fit a fixed bit-pattern) but rather as indices for constructing and traversing other data structures, e.g. lists. Note that unlike lists and booleans, negative sizes are meaningful for integers.

5.2.2 Annotated types and constraints

Table 5.1 extends the basic type system of Section 4.2.2 with size annotations. Algebraic data types are annotated with either a *size variable* ℓ or the symbol $\boldsymbol{\omega}$ indicating the absence of size information. Type variables and the unit type require no size annotation.

As in the system of Chin and Khoo, size annotations are variables and size relations are expressed through constraints; this has the advantage of separating the free algebra of types from the algebra of sizes. For example, checking the equality of the sized types

$$\langle (\mathsf{Int}^i, \mathsf{Int}^j) \to \mathsf{Int}^k, \underbrace{1 + k = i + j \wedge 0 \leq i \wedge 0 \leq j}_{\phi} \rangle$$

$$\langle (\mathsf{Int}^i, \mathsf{Int}^j) \to \mathsf{Int}^k, \underbrace{i + j - 1 = k \wedge 0 \leq j \wedge 0 \leq 1 + k}_{\psi} \rangle$$

reduces to checking the equivalence of constraints ϕ and ψ. This simplifies a type checking or inference algorithm: the non-free algebra of arithmetic can be delegated to a specialised constraint solver, while the inference algorithm deals only with a free algebra of types.

Type variables are syntactical place-holders for type expressions. Because sizes are attached to types, substituting a type variable in will propagate size informa-

τ	$\in$	$\mathbf{Type}_0$	
τ	$::=$	$\alpha \;\mid\; \mathsf{Int}^{z} \;\mid\; \mathsf{D}^{z}\,\vec{\tau}$	zero order types
ν	$::=$	$\tau \;\mid\; \vec{\tau} \to \tau$	first order types
$\vec{\tau}$	$::=$	$(\tau_1, \ldots, \tau_k)$	argument tuple ($k \geq 0$)
z	$::=$	$\ell \;\mid\; \boldsymbol{\omega}$	size annotations
σ	$\in$	$\mathbf{Type}_1$	
σ	$::=$	$\nu \;\mid\; \forall \alpha.\, \sigma'$	type quantification
η	$\in$	$\mathbf{Type}_2$	
η	$::=$	$\langle \sigma, \phi \rangle \;\mid\; \forall \ell.\, \eta'$	size quantification
α	$\in$	$\mathbf{TVar}$	type variables
α	$::=$	$\mathsf{a} \;\mid\; \mathsf{b} \;\mid\; \mathsf{c} \;\mid\; \ldots$	sized type variables
	$\mid$	$\widehat{\mathsf{a}} \;\mid\; \widehat{\mathsf{b}} \;\mid\; \widehat{\mathsf{c}} \;\mid\; \ldots$	unsized type variables
ϕ, ψ	$\in$	$\mathbf{F}$	
ϕ	$::=$	$s_1 \leq s_2 \;\mid\; \phi_1 \wedge \phi_2 \;\mid\; \phi_1 \vee \phi_2 \;\mid\; \exists \ell.\, \phi'$	size constraints
s	$::=$	$n \;\mid\; \ell \;\mid\; n \times s' \;\mid\; s_1 + s_2$	size expressions
ℓ	$\in$	$\mathbf{ZVar}$	
ℓ	$::=$	$\mathsf{i} \;\mid\; \mathsf{j} \;\mid\; \mathsf{k} \;\mid\; \mathsf{n} \;\mid\; \mathsf{m} \;\mid\; \ldots$	size variables

Table 5.1: Syntax of annotated types and size constraints.

tion. For soundness reasons, we will need to disallow size propagation in some circumstances. Therefore, we introduce two kinds of type variables: *unsized type variables* that can only be replaced by types where all size annotations are $\boldsymbol{\omega}$; and *sized type variables* that can be replaced by arbitrary annotated types.

Size constraints

The choice of size constraints must balance expressiveness with effectiveness: constraints must be expressive enough to capture useful size relations, but also simple enough to make automatic manipulation tractable. At the very least the type system should be decidable, i.e. we should be able to algorithmically check if a type derivation is well-formed; this motivates the restriction to a decidable fragment of first-order logic with arithmetic.

Following Chin and Khoo (2001), our size constraints are based on *Presburger* *arithmetic*: first-order logic formulae over the naturals with addition and predicates for equality and 'less-than', but no multiplication. Thus $1 + i = j$ is a Presburger formula over variables i, j, but $i \times j = 4$ is not because of the term $i \times j$. The decidability of this logical theory over the naturals was proved in 1929 by M. Presburger; applications to automated theorem proving go back to Cooper (1972). Presburger formulae can be extended with negative numbers and multiplication by constants and still maintain decidability; computational implementations typically handle these extensions, e.g. the Omega calculator (Pugh 1992).

Although we do not consider equality as a primitive predicate, equations can be expressed as a conjunction of two inequations. We use the shorthand $s_1 = s_2$ for the logically equivalent conjunction $(s_1 \leq s_2) \wedge (s_2 \leq s_1)$. Similarly, we use *True* and *False* as shorthands for universal and unsatisfiable constraints, e.g. $\mathit{True} \equiv (0 \leq 0)$ and $\mathit{False} \equiv (1 \leq 0)$.

While the syntax of our size constraints is a fragment of Presburger arithmetic, the semantics is interpreted over the *rationals* rather than the *integers*. This choice allows performing quantifier elimination by Fourier elimination (Chandru 1993) avoiding the congruence predicates needed for integer solutions (Rabin 1977). We remark that solving for rationals rather than integers sizes yields a larger solution set which is always a sound approximation. In practice, we found that rational solutions give short constraints that compose more easily while still capturing accurate upper and lower bounds on sizes.

Quantified types

Types can be quantified in both type and size variables. We introduce two levels of quantification: *type schemes* σ are (first-order) types quantified over type variables; and *size schemes* η are type schemes quantified over size variables. As is usual in type systems extended with constraints (Mitchell 1984, Jouvelot and Gifford 1991, Nielson et al. 1999), size schemes need to capture *both* the type structure σ and a constraint ϕ.

We use the logic quantifier $\forall$ for both type and size variables, distinguishing the two uses by the kind of variable quantified. We will also abuse notation and write $\forall\vec{\alpha}$ or $\forall\vec{\ell}$ to quantify over sequences of type or size variables.

Free and bound occurrences of variables

The scope of a type quantifier $\forall\alpha.\,\sigma$ is σ; the scope of a size quantifier $\forall\ell.\,\eta$ is η; the scope of the constraint quantifier $\exists\ell.\,\phi$ is ϕ. An occurrence of a type variable α inside the scope of a quantifier $\forall\alpha$ is said to be *bound* by the quantifier (and similarly for occurrences of size variables ℓ in the scope of $\forall\ell$ or $\exists\ell$). An occurrence of a variable that is not bound by any quantifier is said to be *free*. The set of type and annotation variables with free occurrences in t (where t is a quantified type or a size constraint) is given by $\mathrm{FTV}(t)$ and $\mathrm{FZV}(t)$, respectively (see Table 5.2).

Sized type substitutions

We consider an extended notion of syntactical substitution that maps type variables to (sized) types *and* size variables to size annotations (i.e. size variables or $\boldsymbol{\omega}$). Formally, substitutions θ are sequences of mappings from variables to terms,

$$\theta \ ::= \ [\,] \ \mid \ [\alpha \mapsto \tau]\theta \ \mid \ [\ell \mapsto \ell']\theta \ \mid \ [\ell \mapsto \boldsymbol{\omega}]\theta$$

where $[\,]$ is the empty substitution and:

$[\alpha \mapsto \tau]\theta$ is the substitution mapping a type variable α to the type τ;

$[\ell \mapsto \ell']\theta$ is the substitution mapping a size variable ℓ to another ℓ';

$[\ell \mapsto \boldsymbol{\omega}]\theta$ is the substitution mapping a size variable ℓ to $\boldsymbol{\omega}$.

In all the above cases, a substitution $[v \mapsto \cdots]\theta$ acts as θ for variables other than v.

Substitutions extend to (sized) types in the usual way. We will also extend substitutions to constraints; note that type variables do not occur in the latter, so mappings on type variables are not relevant in this case; substituting a size variable by another in a constraint is straightforward. The only remaining case is the substitution of a size variable by $\boldsymbol{\omega}$ in a constraint ϕ; this is defined by existential quantification:

$$([\ell \mapsto \boldsymbol{\omega}]\theta)\, \phi \overset{\text{def}}{=} \theta\, (\exists \ell.\, \phi) \tag{5.5}$$

As usual, the *composition* of two substitutions is written $\theta_1 \circ \theta_2$ and defined as $(\theta_1 \circ \theta_2)\, t \overset{\text{def}}{=} \theta_1\, (\theta_2\, t)$. A substitution θ is *idempotent* if and only if $\theta \circ \theta = \theta$.

An annotated type τ is *unsized* if and only if all size annotations are $\boldsymbol{\omega}$ and all type variables are unsized, i.e. $\mathrm{FZV}(\tau) = \emptyset$ and $\mathrm{FTV}(\tau)$ are unsized. A substitution θ is *proper* if and only if it is idempotent and all unsized variables are mapped to unsized types. It is straightforward to verify that the composition of two proper substitutions is proper.

$$\begin{aligned}
\mathrm{FTV}(\alpha) &= \{\alpha\} \\
\mathrm{FTV}(\mathsf{Int}^{z}) &= \emptyset \\
\mathrm{FTV}(\mathsf{D}^{z}\,\vec{\tau}) &= \mathrm{FTV}(\vec{\tau}) \\
\mathrm{FTV}(\vec{\tau} \to \tau) &= \mathrm{FTV}(\vec{\tau}) \cup \mathrm{FTV}(\tau) \\
\mathrm{FTV}((\tau_1, \ldots, \tau_k)) &= \textstyle\bigcup_{i=1}^{k} \mathrm{FTV}(\tau_i) \\
\mathrm{FTV}(\forall\alpha.\,\sigma) &= \mathrm{FTV}(\sigma) \setminus \{\alpha\} \\
\mathrm{FTV}(\langle\sigma,\, \phi\rangle) &= \mathrm{FTV}(\sigma) \\
\mathrm{FTV}(\forall\ell.\,\eta) &= \mathrm{FTV}(\eta)
\end{aligned}$$

$$\begin{aligned}
\mathrm{FZV}(\alpha) &= \emptyset \\
\mathrm{FZV}(\mathsf{Int}^{\boldsymbol{\omega}}) &= \emptyset \\
\mathrm{FZV}(\mathsf{Int}^{\ell}) &= \{\ell\} \\
\mathrm{FZV}(\mathsf{D}^{\ell}\,\vec{\tau}) &= \{\ell\} \cup \mathrm{FZV}(\vec{\tau}) \\
\mathrm{FZV}(\mathsf{D}^{\boldsymbol{\omega}}\,\vec{\tau}) &= \mathrm{FZV}(\vec{\tau}) \\
\mathrm{FZV}(\vec{\tau} \to \tau) &= \mathrm{FZV}(\vec{\tau}) \cup \mathrm{FZV}(\tau) \\
\mathrm{FZV}((\tau_1, \ldots, \tau_k)) &= \textstyle\bigcup_{i=1}^{k} \mathrm{FZV}(\tau_i) \\
\mathrm{FZV}(\forall\alpha.\,\sigma) &= \mathrm{FZV}(\sigma) \\
\mathrm{FZV}(\langle\sigma,\, \phi\rangle) &= \mathrm{FZV}(\sigma) \cup \mathrm{FZV}(\phi) \\
\mathrm{FZV}(\forall\ell.\,\eta) &= \mathrm{FZV}(\eta) \setminus \{\ell\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{FZV}(\phi_1 \wedge \phi_2) = \mathrm{FZV}(\phi_1 \vee \phi_2) &= \mathrm{FZV}(\phi_1) \cup \mathrm{FZV}(\phi_2) \\
\mathrm{FZV}(s_1 \leq s_2) &= \mathrm{FZV}(s_1) \cup \mathrm{FZV}(s_2) \\
\mathrm{FZV}(\exists\ell.\,\phi) &= \mathrm{FZV}(\phi) \setminus \{\ell\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{FZV}(n) &= \emptyset \\
\mathrm{FZV}(\ell) &= \{\ell\} \\
\mathrm{FZV}(n \times s) &= \mathrm{FZV}(s) \\
\mathrm{FZV}(s_1 + s_2) &= \mathrm{FZV}(s_1) \cup \mathrm{FZV}(s_2)
\end{aligned}$$

Table 5.2: Free type and size variables

$$
\begin{array}{lcl}
\mathcal{V} \models s_1 \le s_2 & \iff & [\![s_1]\!]\,\mathcal{V} \le [\![s_2]\!]\,\mathcal{V} \\
\mathcal{V} \models \phi_1 \wedge \phi_2 & \iff & \mathcal{V} \models \phi_1 \ \wedge\ \mathcal{V} \models \phi_2 \\
\mathcal{V} \models \phi_1 \vee \phi_2 & \iff & \mathcal{V} \models \phi_1 \ \vee\ \mathcal{V} \models \phi_2 \\
\mathcal{V} \models \exists \ell.\, \phi & \iff & \exists r \in \mathbb{Q} : \mathcal{V}[\ell \mapsto r] \models \phi
\end{array}
$$

Table 5.3: Constraint satisfiability relation

Lemma 5.1 *For all size variables ℓ, ℓ' and terms t (i.e. sized types or constraints):*

$$[\ell \mapsto \ell']\, t \;=\; t\,, \quad \textit{if}\ \ell \notin \mathrm{FZV}(t) \tag{5.6}$$

$$\mathrm{FZV}([\ell \mapsto \ell']\, t) \;=\; [\ell \mapsto \ell']\, \mathrm{FZV}(t) \tag{5.7}$$

$$\mathrm{FZV}([\ell \mapsto \boldsymbol{\omega}]\, t) \;=\; \mathrm{FZV}(t) \setminus \{\ell\} \tag{5.8}$$

Proof: By simple induction on the structure of t. □

5.2.3 Semantics of size constraints

Informally, a size constraint represents an approximation to the sizes of runtime data values (this notion will be made formal in Section 5.4). In particular, *True* conveys the least informative approximation corresponding to the absence of any size information; conversely, *False* conveys the most informative approximation, corresponding to statically detecting unreachability (i.e. a run-time error or non-termination).

Valuations and satisfiability

To assign truth values to constraints, we borrow the standard notion of valuation from model theory: a *size valuation* is a function $\mathcal{V} : \mathbf{ZVar} \to \mathbb{Q}$ assigning a rational size to each variable. We use the standard relation of *satisfiability*[2] of a constraint under a valuation written $\mathcal{V} \models \phi$ and defined in Table 5.3.

Entailment

Constraint entailment is the "semantic consequence" relation: $\phi \vDash \psi$ if and only if every valuation satisfying ϕ also satisfies ψ. Informally $\phi \vDash \psi$ means that the size information conveyed by ψ is compatible with (but possibly less precise than)

[2] Note that our notion of satisfiability is more restrictive than the notion of first-order model: we only consider interpretation in the universe $\mathbb{Q}$ and the meaning of constants and operations is always the standard arithmetic interpretation.

that conveyed by ϕ. Entailment is a partial order and in particular *False* $\models \phi$ and $\phi \models$ *True* for all ϕ, i.e. *False* and *True* are the bottom and top elements.

To define entailment formally, we start by defining a pre-order relation and then extend it to an order between equivalence classes.

Definition 5.2 *We say that ϕ* pre-entails *ψ (and write $\phi \models' \psi$) if all valuations satisfying ϕ also satisfy ψ. Formally,*

$$\phi \models' \psi \overset{def}{\iff} \forall \mathcal{V} : \mathbf{ZVar} \to \mathbb{Q}\ (\mathcal{V} \models \phi \implies \mathcal{V} \models \psi)\ .$$

If $\phi \models' \psi$ and $\psi \models' \phi$ then ϕ and ψ are *logically equivalent* and we write $\phi \simeq \psi$. We remark that $\simeq$ is an equivalence relation in $\mathbf{F}$ because $\models'$ is reflexive and transitive. However, $\models'$ is not antisymmetric because equivalent constraints can be syntactically distinct, e.g. $(0 \leq 0) \simeq (0 \leq 1)$. By lifting $\models'$ to the quotient set $\mathbf{F}/\simeq$ we obtain a partial order on equivalence classes of constraints.

Definition 5.3 *$\models$ is a binary relation in $\mathbf{F}/\simeq$ defined by $[\phi_1] \models [\phi_2]$ if and only if $\phi_1 \models' \phi_2$ (where $[\phi]$ is the equivalence class of ϕ with respect to $\simeq$).*

In the remaining presentation we will identify logically equivalent constraints (that is, we consider constraints as representatives of $\mathbf{F}/\simeq$).

Partial order of constraints

We remark that $(\mathbf{F}/\simeq, \models)$ forms a partially ordered set (in fact, a lattice with least upper bound $\vee$ and greatest lower bound $\wedge$). However, $(\mathbf{F}/\simeq, \models)$ is *not* a complete partial order because not all ascending chains have a least upper bound. For example, the circle can be obtained as the limit of a infinite sequence of enclosed convex polyhedra each of which can be represented by a formula in $\mathbf{F}$; yet the circle is not expressible by a formula in $\mathbf{F}$.

Results from first-order logic

The following results regarding existential quantification hold for any first-order logic theory.

Lemma 5.4 *For all formulas ϕ, ψ and variables ℓ, ℓ':*

$$\phi \vDash \psi \implies \phi \vDash \exists \ell.\, \psi \tag{5.9}$$

$$\phi \vDash \psi \implies \exists \ell.\, \phi \vDash \exists \ell.\, \psi \tag{5.10}$$

$$\exists \ell.\, \exists \ell'.\, \phi \simeq \exists \ell'.\, \exists \ell.\, \phi \tag{5.11}$$

$$\exists \ell.\, \phi \simeq \phi\,, \qquad \text{if } \ell \notin \mathrm{FZV}(\phi) \tag{5.12}$$

$$(\exists \ell.\, \phi) \wedge \psi \simeq \exists \ell'.\, ([\ell \mapsto \ell']\phi \wedge \psi)\,, \qquad \text{if } \ell' \notin \mathrm{FZV}(\phi) \cup \mathrm{FZV}(\psi) \tag{5.13}$$

$$\phi \wedge (\exists \ell.\, \psi) \simeq \exists \ell'.\, (\phi \wedge [\ell \mapsto \ell']\psi)\,, \qquad \text{if } \ell' \notin \mathrm{FZV}(\phi) \cup \mathrm{FZV}(\psi) \tag{5.14}$$

$$(\exists \ell.\, \phi) \vee \psi \simeq \exists \ell'.\, ([\ell \mapsto \ell']\phi \vee \psi)\,, \qquad \text{if } \ell' \notin \mathrm{FZV}(\phi) \cup \mathrm{FZV}(\psi) \tag{5.15}$$

$$\phi \vee (\exists \ell.\, \psi) \simeq \exists \ell'.\, (\phi \vee [\ell \mapsto \ell']\psi)\,, \qquad \text{if } \ell' \notin \mathrm{FZV}(\phi) \cup \mathrm{FZV}(\psi) \tag{5.16}$$

Proof: (5.9)–(5.12) follow directly from the definitions of $\vDash$ and $\simeq$; (5.13)–(5.16) are standard in textbooks on mathematical logic, e.g. see Chapter XIII of (Smullyan 1995). □

Result (5.11) permit us to abuse notation and use an existential quantifier with sets of variables $X = \{\ell_1, \ell_2, \ldots, \ell_n\}$, writing $\exists X.\, \phi$ instead of $\exists \ell_1.\, \exists \ell_2.\, \ldots \exists \ell_n.\, \phi$ (where the order among variables is chosen arbitrarily). Repeated applications of results (5.13)–(5.16) allow moving all quantifiers to the front of a formula, i.e. obtaining an equivalent formula in *prenex form.*

Lemma 5.5 *For every formula ϕ, there is a quantifier-free formula ψ such that $\phi \simeq \exists \ell_1.\, \ldots\, \exists \ell_n.\, \psi$.*

Proof: By induction on the structure of ϕ together with (5.13)–(5.16). □

5.3 Sized typing rules

5.3.1 Sized type assumptions

As would be expected, type assumptions must be extended to associate identifiers and constructors to sized types:

$$\Gamma \ ::= \ [\,] \ \mid \ x : \tau, \Gamma \ \mid \ f : \eta, \Gamma \ \mid \ c : \eta, \Gamma$$

Our type system rules will maintain the invariant that zero-order names are associated with simple annotated types while function names and constructors are associated with (type or size) quantified types. In particular, only the function types capture size constraints. The rationale for this distinction is that size relations between zero-order values are expressed by the type judgement, not in individual types.

We employ the same notational conventions as in the underlying type system of Section 4.2.2 regarding concatenation of assumption sequences; we will also use assumptions as (partial) functions, i.e. $\Gamma(id)$ gives the first assumption (if any) for id in the sequence Γ (where id is an identifier or constructor).

5.3.2 Typing judgements for expressions

Table 5.4 presents the sized typing rules for Core Hume expressions as judgements

$$\Gamma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi$$

which state that, under assumptions Γ, expression e admits annotated type τ subject to size constraint ϕ. The rules of Table 5.4 extend the ones for the underlying type system presented in Section 4.2.2:

- The judgement for instantiation $\Gamma \vdash_{\mathsf{INST}} id : \eta$ now handles size as well as type quantifiers. There are two rules for $\forall$ elimination (distinguishing sized and unsized type variables) and two rules for $\forall$ elimination (substituting a variable or $\boldsymbol{\omega}$).

- Rule [*Int*] specifies the most precise size for a literal integer, i.e. the value of integer itself.

- Rules [*FunAp*] and [*ConsAp*] require that the types of arguments and function domain match, including size annotations. The constraint for the application is the conjunction of the constraints for the function and argument; this models a strict semantics: when the argument constraint ϕ' is unsatisfiable (and therefore the argument is $\perp$), the constraint $\phi \wedge \phi'$ for the application is also unsatisfiable.

- Rule [*Case*] specifies the constraint for a case expression as the conjunction of the constraint ϕ_0 for the scrutinised expression and the disjunction of constraints for each alternatives: a "contextual" constraint ϕ'_k associated with any value that matches matches the constructor c_k; and a constraint ϕ_k associated with the expression e_k on the right-hand side of the alternative.

- Rule [*Unsize*] allows forcing the loss of size information by replacing a size annotation by $\boldsymbol{\omega}$.

$$\boxed{\Gamma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi}$$

$$[\mathit{Int}] \quad \Gamma \vdash_{\mathsf{SIZE}} n : \mathsf{Int}^{\ell} \mid \ell = n \qquad \ell \notin \mathrm{FZV}(\Gamma)$$

$$[\mathit{Var}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} x : \langle \tau,\, \phi \rangle}{\Gamma \vdash_{\mathsf{SIZE}} x : \tau \mid \phi} \quad \mathrm{FZV}(\tau) \cap \mathrm{FZV}(\Gamma) = \emptyset$$

$$[\mathit{FunAp}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} f : \langle \vec{\tau} \rightarrow \tau',\, \phi' \rangle \qquad \Gamma \vdash_{\mathsf{SIZE}} \vec{e} : \vec{\tau} \mid \phi}{\Gamma \vdash_{\mathsf{SIZE}} f\ \vec{e} : \tau' \mid \phi \wedge \phi'}$$

$$[\mathit{ConsAp}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} c : \langle \vec{\tau} \rightarrow \tau',\, \phi' \rangle \qquad \Gamma \vdash_{\mathsf{SIZE}} \vec{e} : \vec{\tau} \mid \phi}{\Gamma \vdash_{\mathsf{SIZE}} c\ \vec{e} : \tau' \mid \phi \wedge \phi'}$$

$$[\mathit{Let}] \quad \frac{\Gamma \vdash_{\mathsf{SIZE}} e' : \tau' \mid \phi' \qquad x : \tau',\, \Gamma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi}{\Gamma \vdash_{\mathsf{SIZE}} \mathsf{let}\ x = e'\ \mathsf{in}\ e : \tau \mid \phi \wedge \phi'}$$

$$[\mathit{Case}] \quad \frac{\begin{array}{l} \Gamma \vdash_{\mathsf{SIZE}} e_0 : \tau' \mid \phi_0 \\ \Gamma \vdash_{\mathsf{INST}} c_i : \langle \vec{\tau}''_i \rightarrow \tau',\, \phi'_i \rangle \qquad \vec{x}_i : \vec{\tau}''_i,\ \Gamma \vdash_{\mathsf{SIZE}} e_i : \tau \mid \phi_i \qquad (\forall i) \end{array}}{\Gamma \vdash_{\mathsf{SIZE}} \mathsf{case}\ e_0\ \mathsf{of}\ \{c_i\ \vec{x}_i \rightarrow e_i\}_{i=1}^{n} : \tau \mid \phi_0 \wedge \bigvee_{i=1}^{n}(\phi_i \wedge \phi'_i)}$$

$$[\mathit{Tuple}] \quad \frac{\Gamma \vdash_{\mathsf{SIZE}} e_i : \tau_i \mid \phi_i \qquad (\forall i)}{\Gamma \vdash_{\mathsf{SIZE}} (e_1,\, \ldots,\, e_k) : (\tau_1, \ldots, \tau_k) \mid \bigwedge_{i=1}^{k} \phi_i}$$

$$[\mathit{Unsize}] \quad \frac{\Gamma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi}{\Gamma \vdash_{\mathsf{SIZE}} e : [\ell \mapsto \boldsymbol{\omega}]\, \tau \mid \phi} \quad \ell \notin \mathrm{FZV}(\Gamma)$$

$$[\mathit{Weaken}] \quad \frac{\Gamma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi}{\Gamma \vdash_{\mathsf{SIZE}} e : \tau \mid \psi} \quad \phi \vDash \psi$$

Side conditions:

$$\mathrm{FZV}(\tau) \cap \mathrm{FZV}(\Gamma) = \emptyset \qquad \mathrm{FZV}(\tau) \cap \mathrm{FZV}(\tau') = \emptyset \qquad \mathrm{FZV}(\tau') \cap \mathrm{FZV}(\Gamma) = \emptyset$$

$$\mathrm{FZV}(\tau''_i) \cap \mathrm{FZV}(\Gamma) = \emptyset \qquad \mathrm{FZV}(\tau) \cap \mathrm{FZV}(\tau''_i) = \emptyset \qquad \mathrm{FZV}(\tau') \cap \mathrm{FZV}(\tau''_i) = \emptyset$$

Table 5.4: Sized typing rules for expressions.

$$\boxed{\Gamma \vdash_{\mathsf{INST}} id : \eta}$$

$$[\mathit{Axiom1}] \quad \Gamma \vdash_{\mathsf{INST}} x : \langle \Gamma(x),\ \mathit{True} \rangle$$

$$[\mathit{Axiom2}] \quad \Gamma \vdash_{\mathsf{INST}} id : \Gamma(id) \qquad id \text{ is a constructor or function name}$$

$$[\mathit{Rename}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} id : \langle \sigma,\ \phi \rangle}{\Gamma \vdash_{\mathsf{INST}} id : \langle [\ell \mapsto \ell']\, \sigma,\ \phi \wedge \ell = \ell' \rangle} \qquad \ell' \notin \mathrm{FZV}(\Gamma) \cup \mathrm{FZV}(\sigma)$$

$$[\mathit{Elim}\forall\mathit{1}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} id : \langle \forall \alpha.\, \sigma,\ \phi \rangle}{\Gamma \vdash_{\mathsf{INST}} id : \langle [\alpha \mapsto \tau]\, \sigma,\ \phi \rangle} \qquad \alpha \text{ is sized}$$

$$[\mathit{Elim}\forall\mathit{2}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} id : \langle \forall \widehat{\alpha}.\, \sigma,\ \phi \rangle}{\Gamma \vdash_{\mathsf{INST}} id : \langle [\widehat{\alpha} \mapsto \tau]\, \sigma,\ \phi \rangle} \qquad \widehat{\alpha},\ \mathrm{FTV}(\tau) \text{ are unsized and } \mathrm{FZV}(\tau) = \emptyset$$

$$[\mathit{Elim}\forall\mathit{3}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} id : \forall \ell.\, \eta}{\Gamma \vdash_{\mathsf{INST}} id : [\ell \mapsto \ell']\, \eta}$$

$$[\mathit{Elim}\forall\mathit{4}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} id : \forall \ell.\, \eta}{\Gamma \vdash_{\mathsf{INST}} id : [\ell \mapsto \boldsymbol{\omega}]\, \eta}$$

Table 5.5: Sized typing rules for instantiation of assumptions.

$$\boxed{\Gamma \vdash_{\mathsf{SIZE}} \mathit{decl} : \eta}$$

$$[\mathit{Fun}] \quad \frac{x_1 : \tau_1, \ldots, x_k : \tau_k,\ \Gamma \vdash_{\mathsf{SIZE}} e : \tau_{k+1} \mid \phi}{\Gamma \vdash_{\mathsf{SIZE}} \mathsf{let}\ f\ (x_1, \ldots, x_k) = e\ :\ \forall \vec{\ell}.\, \langle \forall \vec{\alpha}.\, (\tau_1, \ldots, \tau_k) \to \tau_{k+1},\ \phi \rangle}$$

$$[\mathit{Rec}] \quad \frac{x_1 : \tau_1,\ \ldots,\ x_k : \tau_k,\ f : \forall \vec{\ell}.\, \langle (\tau_1, \ldots, \tau_k) \to \tau_{k+1},\ \phi \rangle,\ \Gamma \vdash_{\mathsf{SIZE}} e : \tau_{k+1} \mid \phi}{\Gamma \vdash_{\mathsf{SIZE}} \mathsf{letrec}\ f\ (x_1, \ldots, x_k) = e\ :\ \forall \vec{\ell}.\, \langle \forall \vec{\alpha}.\, (\tau_1, \ldots, \tau_k) \to \tau_{k+1},\ \phi \rangle}$$

where $\vec{\alpha} = \left(\bigcup_{i=1}^{k+1} \mathrm{FTV}(\tau_i) \right) \setminus \mathrm{FTV}(\Gamma)$

$\vec{\ell} = \left(\bigcup_{i=1}^{k+1} \mathrm{FZV}(\tau_i) \cup \mathrm{FZV}(\phi) \right) \setminus \mathrm{FZV}(\Gamma)$

Table 5.6: Sized typing rules for function declarations.

- Rule [*Weaken*] allows replacing a constraint ϕ by any consequence ψ. This rule is called subtyping in (Chin and Khoo 2001); however, since it is not strictly a relation between types but rather between constraints, we call it *weakening* instead. Weakening can be employed to simplify size information, e.g. by existentially quantifying intermediate size variables.

5.3.3 Typing judgements for declarations

The typing rules for function declarations are presented in Table 5.5. Declaration judgements have the form $\Gamma \vdash_{\mathsf{SIZE}} decl : \eta$ meaning that function declaration *decl* admits sized type scheme η.

Non-recursive functions

Rule [*Fun*] extends the corresponding rule of Section 4.2.2 to obtain a pair of annotated type and size constraint for a non-recursive function definition. The size constraint for the function is obtained by typing the function body. The annotated type and constraint are quantified in all free size and type variables.

Example 5.6 Consider the function *max* that computes the maximum of two integers,

$$\mathsf{let}\ max(x,y) = \mathsf{case}\ \mathsf{lte}(x,y)\ \mathsf{of}\ \mathsf{True} \to y\ \mid\ \mathsf{False} \to x$$

where lte is the primitive 'less-than-or-equal' operation on integers and True, False are the boolean values. The sized type assumptions for these are:

$$\begin{aligned}\Gamma \overset{\text{def}}{=} {} & \mathsf{True} : \forall k.\, \langle \mathsf{Bool}^k,\ k = 1\rangle, \\ & \mathsf{False} : \forall k.\, \langle \mathsf{Bool}^k,\ k = 0\rangle, \\ & \mathsf{lte} : \forall ijk.\, \langle (\mathsf{Int}^i, \mathsf{Int}^j) \to \mathsf{Bool}^k,\ (i \le j \wedge k = 1) \vee (j + 1 \le i \wedge k = 0)\rangle,\ []\end{aligned}$$

The sized type derivation for *max* is presented in Table 5.7. Note that the application of weakening in (5.20) is valid because $\phi \vDash \exists i'j'k'.\,\phi$. The final size formula $(i \le j \wedge k = j) \vee (j + 1 \le i \wedge k = i)$ is an exact characterisation of the maximum of the two integer sizes i and j. □

Recursive functions

Typing a recursive function declaration requires "guessing" the annotated type and size formula. By rule [*Rec*], this assumption is admissible for recursive function if it is an invariant for the function body (the soundness of this rule will be established

$$\Gamma' \vdash_{\textsf{INST}} \textsf{Ite} : \langle (\textsf{Int}^{i'}, \textsf{Int}^{j'}) \to \textsf{Bool}^{k'},\ (i' \le j' \wedge k' = 1) \vee (j' + 1 \le i' \wedge k' = 0)\rangle \tag{5.17}$$

$$\frac{(5.17) \qquad \dfrac{\Gamma' \vdash_{\textsf{SIZE}} x : \textsf{Int}^{i'} \mid i' = i \qquad \Gamma' \vdash_{\textsf{SIZE}} y : \textsf{Int}^{j'} \mid j' = j}{\Gamma' \vdash_{\textsf{SIZE}} (x, y) : (\textsf{Int}^{i'}, \textsf{Int}^{j'}) \mid i' = i \wedge j' = j}\,[\mathit{Tuple}]}{\begin{array}{l}\Gamma' \vdash_{\textsf{SIZE}} \textsf{Ite}(x, y) : \textsf{Bool}^{k'} \mid i' = i \wedge j' = j \wedge \\ \quad ((i' \le j' \wedge k' = 1) \vee (j' + 1 \le i' \wedge k' = 0))\end{array}}\,[\mathit{FunAp}] \tag{5.18}$$

$$\frac{\begin{array}{c}(5.18) \qquad \Gamma' \vdash_{\textsf{INST}} \textsf{True} : \langle \textsf{Bool}^{k'},\ k' = 1\rangle \qquad \Gamma' \vdash_{\textsf{INST}} \textsf{False} : \langle \textsf{Bool}^{k'},\ k' = 0\rangle \\ \Gamma' \vdash_{\textsf{SIZE}} y : \textsf{Int}^{k} \mid k = j \qquad \Gamma' \vdash_{\textsf{SIZE}} x : \textsf{Int}^{k} \mid k = i\end{array}}{\begin{array}{l}\Gamma' \vdash_{\textsf{SIZE}} \textsf{case Ite}(x, y) \textsf{ of True} \to y \mid \textsf{False} \to x : \textsf{Int}^{k} \mid \\ i' = i \wedge j' = j \wedge ((i' \le j' \wedge k' = 1) \vee (j' + 1 \le i' \wedge k' = 0)) \wedge \\ \qquad ((k' = 1 \wedge k = j) \vee (k' = 0 \wedge k = i))\end{array}}\,[\mathit{Case}] \tag{5.19}$$

$$\frac{(5.19)}{\begin{array}{l}\underbrace{x : \textsf{Int}^{i},\ y : \textsf{Int}^{j},\ \Gamma}_{\Gamma'} \vdash_{\textsf{SIZE}} \textsf{case Ite}(x, y) \textsf{ of True} \to y \mid \textsf{False} \to x : \textsf{Int}^{k} \mid \\ \qquad (i \le j \wedge k = j) \vee (j + 1 \le i \wedge k = i)\end{array}}\,[\mathit{Weaken}]\ (^{*}) \tag{5.20}$$

$$\frac{(5.20)}{\begin{array}{l}\Gamma \vdash_{\textsf{SIZE}} \textsf{let } \mathit{max}(x, y) = \textsf{case Ite}(x, y) \textsf{ of True} \to y \mid \textsf{False} \to x \\ \quad : \forall ijk.\ \langle (\textsf{Int}^{i}, \textsf{Int}^{j}) \to \textsf{Int}^{k},\ (i \le j \wedge k = j) \vee (j + 1 \le i \wedge k = i)\rangle\end{array}}\,[\mathit{Fun}] \tag{5.21}$$

$$(^{*}) : \exists i'j'k'.\ (i = i' \wedge j = j' \wedge \ldots) \simeq (i \le j \wedge k = j) \wedge (j + 1 \le i \wedge k = i)$$

Table 5.7: Sized type derivation for the maximum of two integers

in Section 5.4). This rule allows checking a sized type, but does not give a method for constructing a size formula. In Section 5.5 we will present algorithmic methods to automatically construct the size formula using fixed point iteration.

Rule [*Rec*] specifies size quantification for the assumption used in typing the function body, i.e. *polymorphic recursion* restricted to size variables. This is required if the function is used more than once: each application context requires a distinct size instance. Unlike polymorphic recursion on type variables, polymorphic recursion on size variables is decidable. Finally, as in the non-recursive case, the result type and formula are quantified in all free size and type variables.

Example 5.7 We show that the list concatenation function

$$\begin{aligned} \mathsf{letrec}\ \mathit{app}\ (xs, ys) \ &= \ \mathsf{case}\ xs\ \mathsf{of} \\ &\qquad \mathsf{Nil} \to ys \\ &\qquad \mid\ \mathsf{Cons}\ (x, xs') \to \mathsf{Cons}\ (x, \mathit{app}\ (xs', ys)) \end{aligned}$$

admits the sized type

$$\forall ijk.\, \langle \forall a.\, (\mathsf{List}^i\, a, \mathsf{List}^j\, a) \to \mathsf{List}^k\, a,\ k = i + j \wedge 0 \leq i \wedge 0 \leq j \rangle \tag{5.22}$$

The type derivation is presented in Table 5.8, where the following abbreviations are used:

$$\begin{aligned} \Gamma_0 &\stackrel{\text{def}}{=} \mathsf{Nil} : \forall i.\, \langle \forall a.\, \mathsf{List}^i\, a,\ i = 0 \rangle, \\ &\qquad \mathsf{Cons} : \forall ij.\, \langle \forall a.\, (a, \mathsf{List}^i\, a) \to \mathsf{List}^j\, a,\ j = 1 + i \wedge 0 \leq i \rangle \\ \Gamma_1 &\stackrel{\text{def}}{=} xs : \mathsf{List}^i\, a,\ ys : \mathsf{List}^j\, a,\ \mathit{app} : (5.22),\ \Gamma_0 \\ \Gamma_2 &\stackrel{\text{def}}{=} x : a,\ xs' : \mathsf{List}^{n'}\, a,\ \Gamma_1 \end{aligned}$$

Note how size polymorphism is used in judgement (5.24) to obtain an instance of the size formula for the application of append. Finally, we remark that the side-condition of weakening in judgement (5.28) is valid because the size formula in (5.28) is a consequence of the one in (5.27) by existentially quantifying variables n, n', n'', m, m', k'. □

5.4 Soundness

Informally, the soundness result for the size typing judgements of Section 5.2 will state that if we derive a size constraint for an expression, then the size of the expression's value is approximated by the size constraint.

$$\frac{\Gamma_2 \vdash_{\mathsf{SIZE}} xs' : \mathsf{List}^{n''}\, a \mid n'' = n' \qquad \Gamma_2 \vdash_{\mathsf{SIZE}} ys : \mathsf{List}^{m'}\, a \mid m' = j}{\Gamma_2 \vdash_{\mathsf{SIZE}} (xs', ys) : (\mathsf{List}^{n''}\, a, \mathsf{List}^{m'}\, a) \mid n'' = n' \wedge m' = j}\;[\mathit{Tuple}] \tag{5.23}$$

$$\frac{\begin{array}{l}\Gamma_2 \vdash_{\mathsf{INST}} app : \langle(\mathsf{List}^{n''}\, a, \mathsf{List}^{m'}\, a) \to \mathsf{List}^{k'}\, a,\; n'' + m' = k' \wedge \\ \qquad\qquad\qquad 0 \leq n'' \wedge 0 \leq m'\rangle \qquad (5.23)\end{array}}{\begin{array}{r}\Gamma_2 \vdash_{\mathsf{SIZE}} app(xs', ys) : \mathsf{List}^{k'}\, a \mid n'' = n' \wedge m' = j \wedge n'' + m' = k' \wedge \\ 0 \leq n'' \wedge 0 \leq m'\end{array}}\;[\mathit{FunAp}] \tag{5.24}$$

$$\frac{\Gamma_2 \vdash_{\mathsf{SIZE}} x : a \mid \mathit{True} \qquad (5.24)}{\begin{array}{r}\Gamma_2 \vdash_{\mathsf{SIZE}} (x, app(xs', ys)) : (a, \mathsf{List}^{k'}\, a) \mid n'' = n' \wedge m' = j \wedge n'' + m' = k' \wedge \\ 0 \leq n'' \wedge 0 \leq m'\end{array}}\;[\mathit{Tuple}] \tag{5.25}$$

$$\frac{\Gamma_2 \vdash_{\mathsf{INST}} \mathsf{Cons} : \langle(a, \mathsf{List}^{k'}\, a) \to \mathsf{List}^{k}\, a,\; k = 1 + k' \wedge 0 \leq k'\rangle \qquad (5.25)}{\begin{array}{r}\Gamma_2 \vdash_{\mathsf{SIZE}} \mathsf{Cons}(x, app(xs', ys)) : \mathsf{List}^{k}\, a \mid n'' = n' \wedge m' = j \wedge n'' + m' = k' \\ \wedge 0 \leq n'' \wedge 0 \leq m' \wedge k = 1 + k' \wedge 0 \leq k'\end{array}}\;[\mathit{ConsAp}] \tag{5.26}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\mathsf{SIZE}} xs : \mathsf{List}^{n}\, a \mid n = i \\ \Gamma_1 \vdash_{\mathsf{INST}} \mathsf{Nil} : \langle \mathsf{List}^{n}\, a,\; n = 0\rangle \\ \Gamma_1 \vdash_{\mathsf{INST}} \mathsf{Cons} : \langle(a, \mathsf{List}^{n'}\, a) \to \mathsf{List}^{n}\, a,\; n = 1 + n' \wedge 0 \leq n'\rangle \\ \Gamma_1 \vdash_{\mathsf{SIZE}} ys : \mathsf{List}^{k}\, a \mid k = j \qquad (5.26)\end{array}}{\begin{array}{l}\Gamma_1 \vdash_{\mathsf{SIZE}} \mathsf{case}\ xs\ \mathsf{of}\ \cdots : \mathsf{List}^{k}\, a \mid n = i \wedge ((n = 0 \wedge k = j) \vee \\ (n = 1 + n' \wedge 0 \leq n' \wedge n'' = n' \wedge m' = j \wedge n'' + m' = k' \wedge \\ \qquad\qquad 0 \leq n'' \wedge 0 \leq m' \wedge k = 1 + k' \wedge 0 \leq k'))\end{array}}\;[\mathit{Case}] \tag{5.27}$$

$$\frac{(5.27)}{\Gamma_1 \vdash_{\mathsf{SIZE}} \mathsf{case}\ xs\ \mathsf{of}\ \cdots : \mathsf{List}^{k}\, a \mid k = i + j \wedge 0 \leq i \wedge 0 \leq j}\;[\mathit{Weaken}] \tag{5.28}$$

$$\frac{(5.28)}{\begin{array}{r}\Gamma_0 \vdash_{\mathsf{SIZE}} \mathsf{letrec}\ app\ (xs, ys) = \cdots : \forall ijk.\, \langle \forall a.\, (\mathsf{List}^{i}\, a, \mathsf{List}^{j}\, a) \to \mathsf{List}^{k}\, a, \\ k = i + j \wedge 0 \leq i \wedge 0 \leq j\rangle\end{array}}\;[\mathit{Rec}] \tag{5.29}$$

$\Gamma_0 \overset{\text{def}}{=} \mathsf{Nil} : \forall i.\, \langle \forall a.\, \mathsf{List}^{i}\, a,\; i = 0\rangle,\; \mathsf{Cons} : \forall ij.\, \langle \forall a.\, (a, \mathsf{List}^{i}\, a) \to \mathsf{List}^{j}\, a,\; j = 1 + i \wedge 0 \leq i\rangle$

$\Gamma_1 \overset{\text{def}}{=} xs : \mathsf{List}^{i}\, a,\; ys : \mathsf{List}^{j}\, a,\; app : (5.22),\; \Gamma_0$

$\Gamma_2 \overset{\text{def}}{=} x : a,\; xs' : \mathsf{List}^{n'}\, a,\; \Gamma_1$

Table 5.8: Sized type derivation for list append

$$
\begin{array}{rcl}
\mathcal{T} & : & \mathbf{Type}_1 \to \mathbf{TEnv} \to \mathbb{T}_0 \cup \mathbb{T}_1 \\
\mathcal{T}[\![\, \alpha \,]\!]\, \chi & = & \chi(\alpha) \\
\mathcal{T}[\![\, \mathsf{Int}^z \,]\!]\, \chi & = & \mathbb{Z}_\perp \\
\mathcal{T}[\![\, \mathsf{D}^z\, \vec{\tau} \,]\!]\, \chi & = & \Psi_{\mathsf{D}}\ (\mathcal{T}[\![\, \vec{\tau} \,]\!]\, \chi) \\
\mathcal{T}[\![\, \vec{\tau} \to \tau' \,]\!]\, \chi & = & \mathcal{T}[\![\, \vec{\tau} \,]\!]\, \chi \boxright \mathcal{T}[\![\, \tau' \,]\!]\, \chi \\
\mathcal{T}[\![\, \forall \alpha.\, \sigma \,]\!]\, \chi & = & \displaystyle\bigcap_{I \in \mathbb{T}_0} \mathcal{T}[\![\, \sigma \,]\!]\, \chi[\alpha \mapsto I] \\
\mathcal{T}[\![\, () \,]\!]\, \chi & = & \{\mathbf{u}\}_\perp \\
\mathcal{T}[\![\, (\tau_1, \ldots, \tau_n) \,]\!]\, \chi & = & \mathcal{T}[\![\, \tau_1 \,]\!]\, \chi \boxtimes \cdots \boxtimes \mathcal{T}[\![\, \tau_n \,]\!]\, \chi \\
\\
\boxtimes, \boxplus & : & \mathbb{T}_0 \times \mathbb{T}_0 \to \mathbb{T}_0 \\
A \boxtimes B & \overset{\text{def}}{=} & \{\, \lfloor \langle a,\, b \rangle \rfloor \in \mathbf{V}_\perp : \lfloor a \rfloor \in A \wedge \lfloor b \rfloor \in B \,\} \cup \{\perp\} \\
A \boxplus B & \overset{\text{def}}{=} & A \cup B \\
\boxright & : & \mathbb{T}_0 \times \mathbb{T}_0 \to \mathbb{T}_1 \\
A \boxright B & \overset{\text{def}}{=} & \{F \in [\mathbf{V} \to \mathbf{V}_\perp] : F^*(A) \subseteq B\}
\end{array}
$$

Table 5.9: Unsized type semantics.

More formally, to establish the soundness of the size typing rules of Tables 5.4 and 5.5 we will first define an unsized type semantics based on *ideals*, i.e. subsets of CPOs closed for least upper bounds (see Section A.1). This is essentially a first-order restriction of a standard denotational type semantics such as (MacQueen and Sethi 1982). Zero order types will be associated with ideals of $\mathbf{V}_\perp$ and first order types with ideals of $[\mathbf{V} \to \mathbf{V}_\perp]$.

We then refine this semantics with size information: the inhabitants of a sized type $\langle \sigma,\, \phi \rangle$ are inhabitants of the unsized semantics of σ whose size satisfies the constraint ϕ. The crucial result is that such a refinement is still closed for least upper bounds, i.e. it still defines ideals (Lemma 5.16).

5.4.1 Unsized type semantics

Definition 5.8 *The set $\mathbb{T}_0$ of zero order types is the set of ideals of $\mathbf{V}_\perp$; the set $\mathbb{T}_1$ of first order types is the set of ideals of $[\mathbf{V} \to \mathbf{V}_\perp]$.*

$$
\begin{aligned}
\mathbb{T}_0 &\overset{\textit{def}}{=} \{I \subseteq \mathbf{V}_\perp : I \textit{ is an ideal of } \mathbf{V}_\perp\} \\
\mathbb{T}_1 &\overset{\textit{def}}{=} \{I \subseteq [\mathbf{V} \to \mathbf{V}_\perp] : I \textit{ is an ideal of } [\mathbf{V} \to \mathbf{V}_\perp]\}
\end{aligned}
$$

In order to present the type semantics in a compositional way, we use operators $\boxtimes$, $\boxplus$ and $\boxminus$ to construct the semantics for product, sum and function types (MacQueen and Sethi 1982). To assign meaning to free type variables in type expressions, we use *type environments* χ mapping type variables to type denotations, i.e. elements of $\mathbb{T}_0$. The empty type environment χ_0 maps every variable to the bottom element of $\mathbb{T}_0$. As is standard in type systems with Hindley-Milner polymorphism, the semantics of quantified types is the intersection of ideals.

The unsized semantics of types is given by the function $\mathcal{T}$ defined in Table 5.9. The equation defining the semantics of an algebraic data types makes use of a type constructor function

$$\Psi_{\mathsf{D}} : \mathbb{T}_0 \to \mathbb{T}_0$$

for each data type D. The constructor functions can be derived from the data declarations by translation into sums of products; recursive data types are defined as the least fixed point of a function on $\mathbb{T}_0 \to \mathbb{T}_0$. For example, for the types of booleans and lists

$$\begin{aligned} \mathsf{data\ Bool}\ () &= \mathsf{True}\ ()\ \mid\ \mathsf{False}\ () \\ \mathsf{data\ List}\ a &= \mathsf{Nil}\ ()\ \mid\ \mathsf{Cons}\ (a, \mathsf{List}\ a) \end{aligned}$$

the type constructor functions are:

$$\begin{aligned} \Psi_{\mathsf{Bool}} &= \lambda I.\, (\{\mathsf{True}\}_\perp \boxtimes \{\mathbf{u}\}_\perp) \boxplus (\{\mathsf{False}\}_\perp \boxtimes \{\mathbf{u}\}_\perp) \\ &= \lambda I.\, \{\langle \mathsf{True},\, \mathbf{u}\rangle,\, \langle \mathsf{False},\, \mathbf{u}\rangle\}_\perp \\ \Psi_{\mathsf{List}} &= \lambda X.\, \mathit{fix}(\lambda L.\, (\{\mathsf{Nil}\}_\perp \boxtimes \{\mathbf{u}\}_\perp) \boxplus (\{\mathsf{Cons}\}_\perp \boxtimes X \boxtimes L)) \\ &= \lambda X.\, \mathit{fix}(\lambda L.\, \{\langle \mathsf{Nil},\, \mathbf{u}\rangle\}_\perp \boxplus (\{\mathsf{Cons}\}_\perp \boxtimes X \boxtimes L)) \end{aligned}$$

5.4.2 Sized type semantics

To formulate the size correctness of our type system we need to establish an inhabitance relation between the denotations of Section 4.2.3 and the sized types of Section 5.2.

A first problem is that a denotation will admit many sized types: if v inhabits $\langle\sigma,\, \phi\rangle$, then it should also inhabit $\langle\sigma,\, \psi\rangle$ whenever $\phi \vDash \psi$. The approach followed by Chin and Khoo (2001) is to define a "size" function $\mathcal{S}(v :: \sigma)$ that takes a data value v and an annotated type σ and yields the most precise constraint describing the size of the data. Type inhabitance is then defined by constraint entailment: v inhabits $\langle\sigma,\, \phi\rangle$ if $\mathcal{S}(v :: \sigma) \vDash \phi$.

Chin and Khoo define the size of a function $F \in [\mathbf{V} \to \mathbf{V}_\perp]$ as the infinite conjunction of all constraints that are entailed by the function's input/output size relation. Their formal definition is (slightly adapted to our notation):

$$\begin{aligned}\mathcal{S}(F :: \tau_1 \to \tau_2) \overset{\text{def}}{=} \bigwedge\{\phi \in \mathbf{F} : \forall v_1 :: \tau_1\ \forall v_2 :: \tau_2 \\ v_2 = F(v_1) \text{ implies } (\mathcal{S}(v_1 :: \tau_1) \wedge \mathcal{S}(v_2 :: \tau_2)) \vDash \phi\}\end{aligned} \tag{5.30}$$

However, we remark that the equation (5.30) does not always define a formula because the partial order $\vDash$ is incomplete. For example, the size of $F \equiv \lambda n.\, n^2 \in [\mathbb{Z} \to \mathbb{Z}_\perp]$ is undefined because there are infinite descending chains of piecewise-linear approximations to $\{(n, n^2) : n \in \mathbb{Z}\}$ but no "best" approximation (c.f. Section 5.2.3). In fact, (5.30) will be undefined for *any* function which exhibits non-linear size relations; this technical problem invalidates Chin and Khoo's soundness proof.

Our approach to establish the soundness of size approximations is to define a size function for zero-order values (for which the "best" size is well-defined). For function values, we define the approximation relation as a type semantics that imposes the entailment relation between the function's input/output size relation and a size constraint.[3]

Size function

The size function for a value $v \in \mathbf{V}_\perp$ with respect to a annotated type τ is defined in Table 5.10.

We remark that our notion of size is parametrized by the type assumptions for data constructors; consequently, our size function must be defined with respect to the later. This is generalizes the work of Chin and Khoo (2001), where the notion of size is defined only for booleans, integers and lists.

We will therefore consider a restricted form of the type assumptions defined in Section 5.3:

$$\Sigma \ ::= \ [] \ \ | \ \ c : \eta,\, \Sigma$$

Each entry $c : \eta$ associates a constructor c with a size quantified type η (including a size constraint).

Constructor consistency

In order to obtain sound size derivations, we have to impose the precondition that the size constraints in Σ are *consistent* with the denotational semantics. Informally,

[3] This is analogous to establishing correctness of abstract interpretations with an incomplete abstract domain; in this setting the soundness of approximations is expressed using a concretisation function alone (i.e. there is no adjoint abstraction function) (Cousot and Cousot 1992*a*,*b*).

$$
\begin{aligned}
\mathcal{S}_\Sigma(\bot :: \tau) &\overset{\text{def}}{=} \mathit{False} \\
\mathcal{S}_\Sigma(\lfloor n \rfloor :: \mathsf{Int}^{\ell}) &\overset{\text{def}}{=} \ell = n \\
\mathcal{S}_\Sigma(\lfloor n \rfloor :: \mathsf{Int}^{\boldsymbol{\omega}}) &\overset{\text{def}}{=} \mathit{True} \\
\mathcal{S}_\Sigma(\lfloor v \rfloor :: \alpha) &\overset{\text{def}}{=} \mathit{True} \\
\mathcal{S}_\Sigma(\lfloor \langle c,\, v\rangle \rfloor :: \tau') &\overset{\text{def}}{=} \exists X.\, (\phi' \wedge \mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}))\,, \\
&\quad \text{where} \begin{cases} \Sigma \vdash_{\mathsf{INST}} c : \langle \vec{\tau} \to \tau',\, \phi' \rangle \\ \mathrm{FZV}(\vec{\tau}) \cap \mathrm{FZV}(\tau') = \emptyset \\ X = \mathrm{FZV}(\vec{\tau}) \end{cases} \\
\mathcal{S}_\Sigma(\lfloor \langle v_1, \ldots, v_n \rangle \rfloor :: (\tau_1, \ldots, \tau_n)) &\overset{\text{def}}{=} \bigwedge_{i=1}^{n} \mathcal{S}_\Sigma(\lfloor v_i \rfloor :: \tau_i)
\end{aligned}
$$

Table 5.10: Size function for zero-order values and tuples.

consistency of Σ states that that if v is non-bottom then also the constructed value $\langle c,\, v\rangle$ is non-bottom and should therefore have a satisfiable size constraint.

Definition 5.9 *We say Σ is* consistent *if and only for all c and all $v \in \mathbf{V}$ such that $\Sigma \vdash_{\mathsf{INST}} c : \langle \vec{\tau} \to \tau',\, \phi' \rangle$ and $\mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}) = \phi$, if ϕ is satisfiable then $\phi \wedge \phi'$ is also satisfiable.*

Size semantics for functions

The size semantics for function types is defined by extension: a formula ϕ is a sound approximation for F if and only if it ϕ approximates the input/output size relation of F, i.e. the sizes of pairs $(v,\, F(v))$.

Definition 5.10 *The semantics of first order sized types is given by*

$$
\begin{aligned}
\mathcal{T}_\Sigma[\![\langle \forall \vec{\alpha}.\, \vec{\tau} \to \tau',\, \phi \rangle]\!] \overset{\mathit{def}}{=} \{\, F \in [\mathbf{V} \to \mathbf{V}_\bot] \ :\ & F \in \mathcal{T}[\![\, \forall \vec{\alpha}.\, \vec{\tau} \to \tau' \,]\!]\chi_0 \ \mathit{and} \\
& \forall v \in \mathbf{V} \quad \mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(F(v) :: \tau') \vDash \phi \,\}
\end{aligned}
$$

and χ_0 is the empty type environment.

Note that, because there are no free type variables in a quantified first order type, $\mathcal{T}_\Sigma$ does not need a type environment argument.

Example 5.11 Consider an inductive data type for natural numbers augmented with a size measure for magnitude of the numbers:

$$\begin{array}{rcll} \mathsf{data\ Nat}^n & = & \mathsf{Zero} & \{n = 0\} \\ & \mid & \mathsf{Succ\ Nat}^k & \{n = 1 + k\} \end{array} \tag{5.31}$$

and consider the primitive recursive addition function:

$$\begin{array}{rl} \mathsf{letrec}\ \mathit{plus}\ (x,\, y) & = \mathsf{case}\ x\ \mathsf{of} \\ & \quad \mathsf{Zero} \to y \\ & \mid \mathsf{Succ}\ x' \to \mathsf{Succ}\ (\mathit{plus}\ (x', y)) \end{array} \tag{5.32}$$

Designate by $\ulcorner n \urcorner$ the denotation of a natural n, i.e. n applications of Succ followed by one application of Zero:

$$\ulcorner 0 \urcorner \stackrel{\text{def}}{=} \langle \mathsf{Zero},\, \mathbf{u} \rangle$$
$$\ulcorner 1 + n \urcorner \stackrel{\text{def}}{=} \langle \mathsf{Succ},\, \ulcorner n \urcorner \rangle$$

Let $\varphi_{plus} = \mathcal{D}[\![\, (5.32)\,]\!]\, \varphi_0$, i.e. φ_{plus} is the denotational semantics of *plus*. We show that

$$\varphi_{plus} \in \mathcal{T}_\Sigma[\![\forall ijk.\, \langle (\mathsf{Nat}^i, \mathsf{Nat}^j) \to \mathsf{Nat}^k,\, i + j = k \rangle]\!] \tag{5.33}$$

where Σ are the type assumptions associated with the declaration (5.31).

By the soundness of the underlying type system we will assume $\varphi_{plus} \in \mathcal{T}[\![\, (\mathsf{Nat}^i, \mathsf{Nat}^j) \to \mathsf{Nat}^k\,]\!]$. It remains to be proved that

$$\begin{array}{r} \mathcal{S}_\Sigma(\lfloor \langle \ulcorner n \urcorner,\, \ulcorner m \urcorner \rangle \rfloor :: (\mathsf{Nat}^i, \mathsf{Nat}^j)) \wedge \mathcal{S}_\Sigma(\varphi_{plus}\, \langle \ulcorner n \urcorner,\, \ulcorner m \urcorner \rangle :: \mathsf{Nat}^k) \vDash i + j = k \\ (\forall n \in \mathbb{N})\ (\forall m \in \mathbb{N}) \end{array} \tag{5.34}$$

By the definition of $\mathcal{S}$ and Σ:

$$\begin{aligned} \mathcal{S}_\Sigma(\lfloor \langle \ulcorner n \urcorner,\, \ulcorner m \urcorner \rangle \rfloor :: (\mathsf{Nat}^i, \mathsf{Nat}^j)) &= \mathcal{S}_\Sigma(\lfloor \ulcorner n \urcorner \rfloor :: \mathsf{Nat}^i) \wedge \mathcal{S}_\Sigma(\lfloor \ulcorner m \urcorner \rfloor :: \mathsf{Nat}^j) \\ &\simeq i = n \wedge j = m \end{aligned}$$

By the definition of *plus*:

$$\varphi_{plus}\, \langle \ulcorner n \urcorner,\, \ulcorner m \urcorner \rangle = \lfloor \ulcorner n + m \urcorner \rfloor$$

which implies

$$\mathcal{S}_\Sigma(\varphi_{plus}\, \langle \ulcorner n \urcorner,\, \ulcorner m \urcorner \rangle :: \mathsf{Nat}^k) = \mathcal{S}_\Sigma(\lfloor \ulcorner n + m \urcorner \rfloor :: \mathsf{Nat}^k) \simeq k = n + m$$

Replacing both results in our proof obligation (5.34) we obtain a valid entailment between constraints:

$$i = n \wedge j = m \wedge k = n + m \vDash i + j = k \quad (\forall n \in \mathbb{N})\ (\forall m \in \mathbb{N})$$

This concludes the proof of (5.33). □

5.4.3 Preliminary results

The next results show some syntactical properties regarding the size function, namely that only the size variables occurring in a type are free in the size constraint; and that we can commute substitutions and the size function.

Lemma 5.12 $\mathrm{FZV}(\mathcal{S}_\Sigma(v :: \sigma)) \subseteq \mathrm{FZV}(\sigma)$.

Lemma 5.13 $\mathcal{S}_\Sigma(v :: [\ell \mapsto \ell']\,\sigma) \simeq [\ell \mapsto \ell']\,\mathcal{S}_\Sigma(v :: \sigma)$.

Lemma 5.14 $\mathcal{S}_\Sigma(v :: [\ell \mapsto \boldsymbol{\omega}]\,\sigma) \simeq \exists \ell.\, \mathcal{S}_\Sigma(v :: \sigma)$.

Proof of Lemmas 5.12, 5.13 and 5.14: By induction on the structure of σ. □

Intuitively, if a zero order value is not bottom then the size function expresses its "best" size as a formula. The next result shows that this constraint is satisfiable provided that the constructor assumptions are *consistent* (Definition 5.9). This precondition will be needed in the soundness proofs of the typing system.

Lemma 5.15 *If Σ is consistent, $v \in \mathcal{T}[\![\, \sigma \,]\!]\chi_0$, $v \neq \bot$ and each size variable in σ occurs only once, then $\mathcal{S}_\Sigma(v :: \sigma)$ is satisfiable.*

Proof: By induction on the structure of the type σ we construct a valuation for $\mathcal{S}_\Sigma(v :: \sigma)$. The assumption that size variables occur only once is needed to combine valuations of sub-formulas. □

5.4.4 Inclusiveness

We are now ready to prove one further preliminary result, namely that the sized semantics of functional types is *inclusive* (Winskel 1993), i.e. it is closed for least upper bounds of ascending chains. This property is called *admissibility* in (Manna 1974, Chin and Khoo 2001) and is the basis for proving correctness of the typing rule for recursion.

Lemma 5.16 (Inclusiveness of $\mathcal{T}_\Sigma$) *Let $\{F_i : i \in \mathbb{N}\}$ be an ascending chain, i.e. $F_i \in [\mathbf{V} \to \mathbf{V}_\bot]$ and $F_i \sqsubseteq F_{i+1}$ for all $i \in \mathbb{N}$. If $F_i \in \mathcal{T}_\Sigma[\![\langle \forall \vec{\alpha}.\, \vec{\tau} \to \tau',\, \phi\rangle]\!]$ for all i, then $\bigsqcup_{i \in \mathbb{N}} F_i \in \mathcal{T}_\Sigma[\![\langle \forall \vec{\alpha}.\, \vec{\tau} \to \tau',\, \phi\rangle]\!]$.*

Proof: Designate $F = \bigsqcup_{i \in \mathbb{N}} F_i$; we have $F \in \mathcal{T}[\![\, \forall \vec{\alpha}.\, \vec{\tau} \to \tau' \,]\!]\chi_0$ because $\mathcal{T}[\![\, \forall \vec{\alpha}.\, \vec{\tau} \to \tau' \,]\!]\chi_0$ is an ideal and therefore closed for least upper bounds of ascending chains. To prove $F \in \mathcal{T}_\Sigma[\![\langle \forall \vec{\alpha}.\, \vec{\tau} \to \tau',\, \phi\rangle]\!]$ we need to show that

$$\forall v \in \mathbf{V} : \quad \mathcal{S}_\Sigma(v :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(F(v) :: \tau') \vDash \phi$$

Let $v \in \mathbf{V}$ and consider the ascending chain $\{F_i(v) : i \in \mathbb{N}\} \subseteq \mathbf{V}_\perp$. Because $\mathbf{V}_\perp$ is a flat domain, the ascending chain eventually stabilises, i.e. there exists $j \geq 0$ such that $F_j(v) = F_{j+1}(v) = \cdots = F(v)$ and therefore $\mathcal{S}_\Sigma(F(v) :: \tau') \simeq \mathcal{S}_\Sigma(F_j(v) :: \tau')$. Since $F_i \in \mathcal{T}_\Sigma[\![\langle\forall\vec{\alpha}.\, \vec{\tau} \to \tau',\, \phi\rangle]\!]$ for all $i \geq 0$, in particular for $i = j$ we obtain

$$\begin{aligned} & \mathcal{S}_\Sigma(v :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(F_j(v) :: \tau') \vDash \phi \\ \iff\ & \mathcal{S}_\Sigma(v :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(F(v) :: \tau') \vDash \phi \end{aligned}$$

which concludes the proof. □

5.4.5 Preconditions for soundness

To establish the soundness of the typing rules of Section 5.3, we need to formulate some preconditions between the typing assumptions and the denotational semantics. We start by extending the notion of size to denotational environments.

Definition 5.17 *Let ρ be a value environment and Γ be an assumption set. The size of ρ with respect to Γ is:*

$$\mathcal{S}_\Sigma(\rho :: \Gamma) \overset{def}{=} \bigwedge_{i=1}^{n} \mathcal{S}_\Sigma(\rho(x) :: \Gamma(x_i))$$

where $\mathrm{dom}(\rho) = \{x_1, \ldots, x_n\}$.

Definition 5.18 *We say that ρ* satisfies *Γ and write $\rho \models \Gamma$ if and only if for all $x \in \mathrm{dom}(\rho)$ we have $\rho(x) \in \mathcal{T}[\![\, \Gamma(x)\,]\!]\chi_0$.*

Definition 5.19 *We say that φ* satisfies *Γ and write $\varphi \models \Gamma$ if and only if for all $f \in \mathrm{dom}(\varphi)$ we have $\varphi_f \in \mathcal{T}_\Sigma[\![\Gamma(f)]\!]$.*

Inspecting the type rules of Table 5.4 we can see that size constraints in the conclusions of the elimination rules ([*FunAp*], [*ConsAp*], [*Let*] and [*Case*]) can have free occurrences of variables that do not occur free in either the result type or assumptions. However, we can always normalise any derivation $\Gamma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi$ by applying [*Weaken*] to derive $\Gamma \vdash_{\mathsf{SIZE}} e : \tau \mid \exists X.\, \phi$ (where $X = \mathrm{FZV}(\phi) \setminus (\mathrm{FZV}(\tau) \cup \mathrm{FZV}(\Gamma))$ are the intermediate variables). A derivation where all size variables in the formula occur in either the type or assumptions is called *normalised.*

Definition 5.20 *A sized type derivation $\Gamma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi$ is* normalised *if* $\mathrm{FZV}(\phi) \subseteq \mathrm{FZV}(\tau) \cup \mathrm{FZV}(\Gamma)$. *We write $\Gamma \Vdash_{\mathsf{SIZE}} e : \tau \mid \phi$ to indicate that a type derivation is normalised.*

When proving the type soundness results we will assume, without loss of generality, that type derivations for sub-expressions have been normalised. This avoids the possibility of unintended capture of size variables and consequently simplifies the proofs.

5.4.6 Soundness of expression typing

We are now in condition to state and prove the soundness of the typing rules of Table 5.4. Since our semantic notion of size depends on assumptions for constructors, these need to be made explicit in the soundness theorem. We do so by requiring that type assumptions are factored as a concatenation Γ, Σ of two sequences where Γ are the assumptions for identifiers and Σ for constructors. It is always possible to factor any assumption sequence in this way, so no generality is lost.

Informally, the soundness theorem states that if $\Gamma,\, \Sigma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi$ and environments φ and ρ are compatible with Γ and Σ, then ϕ approximates the size of $\mathcal{E}[\![\, e \,]\!]\, \varphi\, \rho$ *and the size of the environment* ρ. The conclusion is more informative than that of a standard type inhabitance result: the type judgement conveys information not just about the sizes of the expression e but also about the sizes of free variables in the environment Γ. As we shall see in the next section, this stronger result is essential to obtain relational size information for function declarations.

Theorem 5.21 (Soundness of expression typing) *If Σ is consistent, $\rho \models \Gamma$, $\varphi \models \Gamma$ and $\Gamma, \Sigma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi$, then $\mathcal{S}_\Sigma(\mathcal{E}[\![\, e \,]\!]\, \varphi\, \rho :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \exists X.\, \phi$, where $X = \mathrm{FZV}(\phi) \setminus (\mathrm{FZV}(\tau) \cup \mathrm{FZV}(\Gamma))$.*

Before proving this theorem, we present an example that illustrates the soundness result.

Example 5.22 Using the type rules of Tables 5.4 and 5.5, we can derive the following judgement

$$xs : \mathsf{List}^n\, a,\, \Sigma \vdash_{\mathsf{SIZE}} \mathsf{case}\ xs\ \mathsf{of}\ \mathsf{Nil} \to \mathsf{True} : \mathsf{Bool}^k \mid \underbrace{n = 0 \wedge k = 1}_{\phi} \tag{5.35}$$

where Σ are the canonical sized type assumptions for lists and boolean constructors:

$$\begin{aligned}
&\mathsf{True} : \forall i.\, \langle \mathsf{Bool}^i,\, i = 1 \rangle \\
&\mathsf{False} : \forall i.\, \langle \mathsf{Bool}^i,\, i = 0 \rangle \\
&\mathsf{Nil} : \forall i.\, \langle \forall a.\, \mathsf{List}^i\, a,\, i = 0 \rangle \\
&\mathsf{Cons} : \forall ij.\, \langle \forall a.\, (a, \mathsf{List}^i\, a) \to \mathsf{List}^j\, a,\, j = 1 + i \rangle
\end{aligned}$$

Let ρ be such that $\rho(xs)$ is a list. Let $v = \mathcal{E}[\![\, \mathsf{case}\ xs\ \mathsf{of}\ \mathsf{Nil} \rightarrow \mathsf{True} \,]\!]\, \varphi_0\, \rho$; then by Theorem 5.21 we have

$$\mathcal{S}_\Sigma(v :: \mathsf{Bool}^k) \wedge \mathcal{S}_\Sigma(\rho(xs) :: \mathsf{List}^n\ a) \vDash k = 1 \wedge n = 0 \tag{5.36}$$

If $v \neq \bot$ and $\rho(xs) \neq \bot$ then from equation (5.36) we can conclude that v is True and $\rho(xs)$ is Nil. This example illustrates how the size judgement conveys information not just about the size of the result but also about the free variables in assumptions. □

Proof of Theorem 5.21: We remark that the result is immediate when $\mathcal{E}[\![\, e \,]\!]\, \varphi\, \rho = \bot$ because $\mathcal{S}_\Sigma(\bot :: \tau) = \mathit{False}$. We prove the case $\mathcal{E}[\![\, e \,]\!]\, \varphi\, \rho \neq \bot$ by induction on the derivation of $\Gamma, \Sigma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi$. We assume, without loss of generality, that any sub-derivations are normalised. The induction hypothesis is then:

> For all sub-derivations $\Gamma', \Sigma \Vdash_{\mathsf{SIZE}} e' : \tau' \mid \phi'$, if $\rho' \models \Gamma'$ and $\mathcal{E}[\![\, e' \,]\!]\, \varphi\, \rho' \neq \bot$, then $\mathcal{S}_\Sigma(\mathcal{E}[\![\, e' \,]\!]\, \varphi\, \rho' :: \tau') \wedge \mathcal{S}_\Sigma(\rho' :: \Gamma') \vDash \phi'$.

We proceed by case analysis on the rule at the root of the derivation.

Rule [*FunAp*]: the type rule hypotheses are:

$$\Gamma, \Sigma \vdash_{\mathsf{INST}} f : \langle \vec{\tau} \rightarrow \tau',\ \phi' \rangle \tag{5.37}$$

$$\Gamma, \Sigma \Vdash_{\mathsf{SIZE}} \vec{e} : \vec{\tau} \mid \phi \tag{5.38}$$

$$\mathrm{FZV}(\vec{\tau}) \cap \mathrm{FZV}(\tau') = \emptyset \tag{5.39}$$

$$\mathrm{FZV}(\vec{\tau}) \cap \mathrm{FZV}(\Gamma) = \emptyset \tag{5.40}$$

One hypothesis for the induction proof is that $\mathcal{E}[\![f\ \vec{e}]\!]\, \varphi\, \rho \neq \bot$; the strict semantics of f implies $\mathcal{E}[\![f\ \vec{e}]\!]\, \varphi\, \rho = \varphi_f(v)$ where $\mathcal{E}[\![\vec{e}]\!]\, \varphi\, \rho = \lfloor v \rfloor \neq \bot$. We can therefore apply the induction hypothesis to (5.38) and obtain

$$\mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \phi \tag{5.41}$$

By (5.37) together with the hypothesis that $\varphi \models \Gamma$ we obtain $\varphi_f \in \mathcal{T}_\Sigma[\![\langle \vec{\tau} \rightarrow \tau',\ \phi' \rangle]\!]$, which implies

$$\mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(\varphi_f(v) :: \tau') \vDash \phi' \tag{5.42}$$

Combining (5.41) and (5.42) by conjunction:

$$\begin{aligned}
&\mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(\varphi_f(v) :: \tau') \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \phi \wedge \phi' \\
\Longrightarrow\ & \exists X.\, (\mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(\varphi_f(v) :: \tau') \wedge \mathcal{S}_\Sigma(\rho :: \Gamma)) \vDash \exists X.\, (\phi \wedge \phi') \\
&\qquad \{\text{existentially quantifying both sides}\} \\
\Longleftrightarrow\ & (\exists X.\, \mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau})) \wedge \mathcal{S}_\Sigma(\varphi_f(v) :: \tau') \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \exists X.\, (\phi \wedge \phi') \\
&\qquad \{\text{by hypothesis (5.39) and (5.40)}\} \\
\Longleftrightarrow\ & \mathit{True} \wedge \mathcal{S}_\Sigma(\varphi_f(v) :: \tau') \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \exists X.\, (\phi \wedge \phi') \\
&\qquad \{\text{because } \exists X.\, \mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}) \simeq \mathit{True}\} \\
\Longleftrightarrow\ & \mathcal{S}_\Sigma(\varphi_f(v) :: \tau') \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \exists X.\, (\phi \wedge \phi')
\end{aligned}$$

The last equation establishes the required result.

Rule $[\mathit{ConsAp}]$: the type rule hypotheses are

$$\Gamma, \Sigma \vdash_{\mathsf{INST}} c : \langle \vec{\tau} \to \tau',\, \phi' \rangle \tag{5.43}$$

$$\Gamma, \Sigma \Vdash_{\mathsf{SIZE}} \vec{e} : \vec{\tau} \mid \phi \tag{5.44}$$

$$\mathrm{FZV}(\vec{\tau}) \cap \mathrm{FZV}(\tau') = \emptyset \tag{5.45}$$

$$\mathrm{FZV}(\vec{\tau}) \cap \mathrm{FZV}(\Gamma) = \emptyset \tag{5.46}$$

We have $\mathcal{E}[\![\, c\ \vec{e}\,]\!]\, \varphi\, \rho \neq \bot$ which implies that $\mathcal{E}[\![\, c\ \vec{e}\,]\!]\, \varphi\, \rho = \lfloor \langle c,\, v \rangle \rfloor$ where $\mathcal{E}[\![\, \vec{e}\,]\!]\, \varphi\, \rho = \lfloor v \rfloor \neq \bot$. We are therefore in a position to apply the induction hypothesis to (5.44) and obtain:

$$\mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \phi \tag{5.47}$$

We also have

$$\mathcal{S}_\Sigma(\lfloor \langle c,\, v \rangle \rfloor :: \tau') \simeq \exists X.\, (\phi' \wedge \mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau})), \quad \text{where } X = \mathrm{FZV}(\vec{\tau}) \tag{5.48}$$

Combining (5.47) by conjunction with ϕ' on both sides yields:

$$\begin{aligned}
&\phi' \wedge \mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \phi' \wedge \phi \\
\Longrightarrow\ & \exists X.\, (\phi' \wedge \mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau}) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma)) \vDash \exists X.\, (\phi' \wedge \phi) \\
&\qquad \{\text{existentially quantifying both sides}\} \\
\Longleftrightarrow\ & \exists X.\, (\phi' \wedge \mathcal{S}_\Sigma(\lfloor v \rfloor :: \vec{\tau})) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \exists X.\, (\phi' \wedge \phi) \\
&\qquad \{\text{by hypothesis (5.46)}\} \\
\Longleftrightarrow\ & \mathcal{S}_\Sigma(\lfloor \langle c,\, v \rangle \rfloor :: \tau') \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \exists X.\, (\phi' \wedge \phi) \\
&\qquad \{\text{by (5.48)}\}
\end{aligned}$$

The last equation is the required result.

Rule $[Case]$: the type rule hypotheses are

$$\Gamma, \Sigma \Vdash_{\mathsf{SIZE}} e_0 : \tau' \mid \phi_0 \tag{5.49}$$

$$\Gamma, \Sigma \vdash_{\mathsf{INST}} c_i : \langle \vec{\tau}_i'' \to \tau',\ \phi_i' \rangle \qquad (1 \leq i \leq n) \tag{5.50}$$

$$\vec{x}_i : \vec{\tau}_i'',\ \Gamma,\ \Sigma \Vdash_{\mathsf{SIZE}} e_i : \tau \mid \phi_i \qquad (1 \leq i \leq n) \tag{5.51}$$

$$\mathrm{FZV}(\tau) \cap \mathrm{FZV}(\tau') = \emptyset \tag{5.52}$$

$$\mathrm{FZV}(\Gamma) \cap \mathrm{FZV}(\tau') = \emptyset \tag{5.53}$$

$$\mathrm{FZV}(\tau) \cap \mathrm{FZV}(\vec{\tau}_i'') = \emptyset \qquad (1 \leq i \leq n) \tag{5.54}$$

$$\mathrm{FZV}(\Gamma) \cap \mathrm{FZV}(\vec{\tau}_i'') = \emptyset \qquad (1 \leq i \leq n) \tag{5.55}$$

Designate $F \overset{\text{def}}{=} \mathcal{A}[\![\, \{c_i\ xs_i \to e_i\}_{i=1}^{n} \,]\!]\, \varphi\, \rho$. By definition of $\mathcal{E}$ and the hypothesis $\mathcal{E}[\![\, e \,]\!]\, \varphi\, \rho \neq \bot$ we get that

$$\mathcal{E}[\![\, \mathsf{case}\ e_0\ \mathsf{of}\ \{c_i\ xs_i \to e_i\}_{i=1}^{n} \,]\!]\, \varphi\, \rho \overset{\text{def}}{=} (\text{let } v \Leftarrow \mathcal{E}[\![\, e_0 \,]\!]\, \varphi\, \rho.\ F(v)) \neq \bot \tag{5.56}$$

Therefore $\mathcal{E}[\![\, e_0 \,]\!]\, \varphi\, \rho = \lfloor v \rfloor \neq \bot$ and $F(v) \neq \bot$. By definition of $\mathcal{A}$, this implies that there exists k and l such that $1 \leq k \leq n$, $1 \leq l$, $v = \langle c_k, v_1', \ldots, v_l' \rangle$ and

$$\begin{aligned} F(v) &= \mathcal{M}[\![\, c_k\ (x_1, \ldots, x_l) \to e_k \,]\!]\, \varphi\, \rho\, \langle v_1', \ldots, v_l' \rangle \\ &= (\lambda \langle v_1, \ldots, v_l \rangle.\ \mathcal{E}[\![\, e_k \,]\!]\, \varphi\, \rho[x_1 \mapsto v_1, \ldots, x_l \mapsto v_l])\ \langle v_1', \ldots, v_l' \rangle \\ &= \mathcal{E}[\![\, e_k \,]\!]\, \varphi\, \rho[x_1 \mapsto v_1', \ldots, x_l \mapsto v_l'] \end{aligned} \tag{5.57}$$

We can apply the induction hypothesis to (5.49) and obtain

$$\mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau') \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \phi_0 \tag{5.58}$$

Hypotheses (5.50) and (5.51) for $i = k$ are

$$\Sigma \vdash_{\mathsf{INST}} c_k : \langle \underbrace{(\tau_1, \ldots, \tau_l)}_{\vec{\tau}_k''} \to \tau',\ \phi_k' \rangle \tag{5.59}$$

$$x_1 : \tau_1, \ldots, x_l : \tau_l,\ \Gamma,\ \Sigma \Vdash_{\mathsf{SIZE}} e_k : \tau \mid \phi_k \tag{5.60}$$

where $\vec{\tau}_k'' = (\tau_1, \ldots, \tau_l)$. Let $X = \mathrm{FZV}(\vec{\tau}_k'') = \mathrm{FZV}(\tau_1) \cup \ldots \cup \mathrm{FZV}(\tau_l)$; by the definition of $\mathcal{S}$ together with (5.59) we have

$$\begin{aligned} \mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau') &= \mathcal{S}_\Sigma(\lfloor \langle c_k, v_1', \ldots, v_l' \rangle \rfloor :: \tau') \\ &\simeq \exists X.\, (\phi_k' \wedge \mathcal{S}_\Sigma(\lfloor \langle v_1', \ldots, v_l' \rangle \rfloor :: \vec{\tau}_k'')) \end{aligned} \tag{5.61}$$

Let $\rho' = \rho[x_1 \mapsto v_1', \ldots, x_l \mapsto v_l']$ and $\Gamma' = x_1 : \tau_1, \ldots, x_l : \tau_l, \Gamma$. We are now in condition to apply the induction hypothesis to (5.60) and obtain:

$$\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho' :: \Gamma') \vDash \phi_k \tag{5.62}$$

But

$$\begin{aligned}\mathcal{S}_\Sigma(\rho' :: \Gamma') &= \mathcal{S}_\Sigma(\rho[x_1 \mapsto v'_1, \ldots, x_l \mapsto v'_l] :: (x_1 : \tau_1, \ldots, x_l : \tau_l, \Gamma))\\ &= \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathcal{S}_\Sigma(\lfloor v'_1 \rfloor :: \tau_1) \wedge \ldots \wedge \mathcal{S}_\Sigma(\lfloor v'_l \rfloor :: \tau_l)\\ &= \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathcal{S}_\Sigma(\lfloor \langle v'_1, \ldots, v'_l \rangle \rfloor :: (\tau_1, \ldots, \tau_l))\\ &= \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathcal{S}_\Sigma(\lfloor \langle v'_1, \ldots, v'_l \rangle \rfloor :: \vec{\tau}''_k) \qquad (5.63)\end{aligned}$$

Substituting (5.63) in (5.62) yields:

$$\begin{aligned}
&\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathcal{S}_\Sigma(\lfloor \langle v'_1, \ldots, v'_l \rangle \rfloor :: \vec{\tau}''_k) \vDash \phi_k\\
\Longrightarrow\; &\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathcal{S}_\Sigma(\lfloor \langle v'_1, \ldots, v'_l \rangle \rfloor :: \vec{\tau}''_k) \wedge \phi'_k \vDash \phi_k \wedge \phi'_k\\
\Longrightarrow\; &\exists X.\, (\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathcal{S}_\Sigma(\lfloor \langle v'_1, \ldots, v'_l \rangle \rfloor :: \vec{\tau}''_k) \wedge \phi'_k) \vDash \exists X.\, (\phi_k \wedge \phi'_k)\\
\Longleftrightarrow\; &\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \exists X.\, (\mathcal{S}_\Sigma(\lfloor \langle v'_1, \ldots, v'_l \rangle \rfloor :: \vec{\tau}''_k) \wedge \phi'_k) \vDash \exists X.\, (\phi_k \wedge \phi'_k)\\
&\qquad \{\text{by (5.54) and (5.55)}\}\\
\Longleftrightarrow\; &\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau') \vDash \exists X.\, (\phi_k \wedge \phi'_k)\\
&\qquad \{\text{by (5.61)}\}\\
\Longrightarrow\; &\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau') \vDash \phi_0 \wedge \exists X.\, (\phi_k \wedge \phi'_k)\\
&\qquad \{\text{by conjunction with (5.58)}\}\\
\Longrightarrow\; &\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau') \vDash \exists X.\, (\phi_0 \wedge \phi_k \wedge \phi'_k)\\
&\qquad \{\text{by (5.54) and (5.55)}\}\\
\Longrightarrow\; &\exists X'.\, (\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau')) \vDash \exists X'.\, \exists X.\, (\phi_0 \wedge \phi_k \wedge \phi'_k)\\
&\qquad \{\text{existentially quantifying over } X' = \mathrm{FZV}(\tau')\}\\
\Longleftrightarrow\; &\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \exists X'.\, \mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau') \vDash \exists X'.\, \exists X.\, (\phi_0 \wedge \phi_k \wedge \phi'_k)\\
&\qquad \{\text{by (5.52) and (5.53)}\}\\
\Longleftrightarrow\; &\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \wedge \mathit{True} \vDash \exists X'.\, \exists X.\, (\phi_0 \wedge \phi_k \wedge \phi'_k)\\
&\qquad \{\text{because } \exists X'.\, \mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau') \simeq \mathit{True} \text{ by Lemma 5.15}\}\\
\Longrightarrow\; &\mathcal{S}_\Sigma(F(v) :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \exists X'.\, \exists X.\, (\phi_0 \wedge \textstyle\bigvee_{i=1}^{n} (\phi_i \wedge \phi'_i))\\
&\qquad \{\text{because } \phi_k \wedge \phi'_k \vDash \textstyle\bigvee_{i=1}^{n} (\phi_i \wedge \phi'_i)\}
\end{aligned}$$

Note that the use of Lemma 5.15 above requires constructor consistency (Definition 5.9). This concludes the proof of rule [*Case*].

Rule [*Unsize*]: let $\lfloor v \rfloor = \mathcal{E}[\![\, e \,]\!]\, \varphi\, \rho \neq \bot$; starting from the induction hypothesis, we get:

$$\begin{aligned}
&\mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \phi \\
\Longrightarrow\ & \exists \ell.\, (\mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma)) \vDash \exists \ell.\, \phi \\
&\qquad \{\text{existentially quantifying both sides over } \ell\} \\
\Longleftrightarrow\ & \exists \ell.\, \mathcal{S}_\Sigma(\lfloor v \rfloor :: \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \exists \ell.\, \phi \\
&\qquad \{\text{by the hypothesis } \ell \notin \mathrm{FZV}(\Gamma)\} \\
\Longleftrightarrow\ & \mathcal{S}_\Sigma(\lfloor v \rfloor :: [\ell \mapsto \boldsymbol{\omega}]\, \tau) \wedge \mathcal{S}_\Sigma(\rho :: \Gamma) \vDash \exists \ell.\, \phi \\
&\qquad \{\text{by Lemma 5.14}\}
\end{aligned}$$

The last equation is the required result.

Rule [*Weaken*]: the result follows directly from the transitivity of $\vDash$.

The remaining cases for [*Int*], [*Var*] and [*Tuple*] are straightforward and we omit them. This concludes the proof of Theorem 5.21. □

5.4.7 Soundness of declaration typing

We can now establish the soundness of the typing rule for function abstractions; the proof splits into two cases according to whether the function is recursive or not. The result for the non-recursive case follows from the application of soundness of expression typing to the function body (Theorem 5.21); for the recursive case, we additionally employ fixed-point induction together with the inclusiveness of the sized type semantics (Lemma 5.16).

Theorem 5.23 (Soundness of declaration typing) *If* Σ *is consistent,* $\varphi \models \Gamma$ *and* $\Gamma, \Sigma \vdash_{\mathit{SIZE}} \mathit{decl} : \eta$*, then* $\mathcal{D}[\![\, \mathit{decl} \,]\!]\, \varphi \in \mathcal{T}_\Sigma[\![\eta]\!]$.

Proof: By simple case analysis, distinguishing recursive and non-recursive function declarations. Note that since all judgements of Table 5.5 comprise a single step, we do not need induction. Again we assume without loss of generality that typing judgements for sub-expressions are normalised.

Case [*Fun*]: The hypothesis for the type rule is

$$x_1 : \tau_1, \ldots, x_k : \tau_k,\ \Gamma,\ \Sigma \Vdash_{\mathsf{SIZE}} e : \tau_{k+1} \mid \phi \tag{5.64}$$

Designate $F = \mathcal{D}[\![\, \mathsf{let}\ f\ (x_1, \ldots, x_k) = e \,]\!]\varphi$; by the definition of $\mathcal{D}$ in Table 4.6, we have

$$F = \lambda\langle v_1, \ldots v_k\rangle.\ \mathcal{E}[\![\, e \,]\!]\, \varphi\, \rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] \tag{5.65}$$

We need to prove $F \in \mathcal{T}_\Sigma[\![\langle \forall \vec{\alpha}.\, (\tau_1, \ldots, \tau_k) \to \tau_{k+1},\, \phi \rangle]\!]$. By Definition 5.10, it suffices to show that

$$\forall v \in \mathbf{V} \quad \mathcal{S}_\Sigma(\lfloor v \rfloor :: (\tau_1, \ldots, \tau_k)) \wedge \mathcal{S}_\Sigma(F(v) :: \tau_{k+1}) \vDash \phi \tag{5.66}$$

To prove (5.66), assume $v \in \mathbf{V}$ and $v = \langle v_1, \ldots, v_k \rangle$. From (5.65) we get $F(v) = \mathcal{E}[\![\, e \,]\!]\, \varphi\, \rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$. It is straightforward to check that

$$\begin{aligned} \varphi &\models x_1 : \tau_1, \ldots, x_k : \tau_k,\, \Gamma \\ \rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] &\models x_1 : \tau_1, \ldots, x_k : \tau_k,\, \Gamma \end{aligned}$$

We can therefore apply Theorem 5.21 to hypothesis (5.64) and obtain

$$\mathcal{S}_\Sigma(\rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] :: (x_1 : \tau_1, \ldots, x_k : \tau_k,\, \Gamma)) \wedge \mathcal{S}_\Sigma(F(v) :: \tau_{k+1}) \vDash \phi \tag{5.67}$$

But

$$\begin{aligned} \mathcal{S}_\Sigma(\rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] :: (x_1 : \tau_1, \ldots, x_k : \tau_k,\, \Gamma)) = \\ = \mathcal{S}_\Sigma(\lfloor v_1 \rfloor :: \tau_1) \wedge \ldots \wedge \mathcal{S}_\Sigma(\lfloor v_k \rfloor :: \tau_k) \\ = \mathcal{S}_\Sigma(\lfloor \langle v_1, \ldots, v_k \rangle \rfloor :: (\tau_1, \ldots, \tau_k)) \\ = \mathcal{S}_\Sigma(\lfloor v \rfloor :: (\tau_1, \ldots, \tau_k)) \end{aligned}$$

Replacing the equality above in (5.67), we get

$$\mathcal{S}_\Sigma(\lfloor v \rfloor :: (\tau_1, \ldots, \tau_k)) \wedge \mathcal{S}_\Sigma(F(v) :: \tau_{k+1}) \vDash \phi$$

which is the desired result.

Case [*Rec*]: The hypothesis for the typing judgement is

$$x_1 : \tau_1,\, \ldots,\, x_k : \tau_k,\, f : \forall \vec{\ell}.\, \langle (\tau_1, \ldots, \tau_k) \to \tau_{k+1},\, \phi \rangle,\, \Gamma,\, \Sigma \Vdash_{\mathsf{SIZE}} e : \tau \mid \phi$$
$$\text{where } \vec{\ell} = \mathrm{FZV}(\phi) \cup \textstyle\bigcup_{i=1}^{k+1} \mathrm{FZV}(\tau_i) \tag{5.68}$$

By the definition of $\mathcal{D}$ in Table 4.6, $\mathcal{D}[\![\, \mathsf{letrec}\ f\ (x_1, \ldots, x_k) = e \,]\!]\, \varphi = \mathit{fix}(\mathcal{F})$, where

$$\mathcal{F} = \lambda F.\ \lambda \langle v_1, \ldots, v_k \rangle.\ \mathcal{E}[\![\, e \,]\!]\ \varphi[f \mapsto F]\ \rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$$

Since $\mathit{fix}(\mathcal{F}) = \bigsqcup_{n \geq 0} \mathcal{F}^n(\bot)$ (where $\bot$ is the least element of $[\mathbf{V} \to \mathbf{V}_\bot]$) it is sufficient to prove

$$\bigsqcup_{n \geq 0} \mathcal{F}^n(\bot) \in \mathcal{T}_\Sigma[\![\langle \forall \vec{\alpha}.\, (\tau_1, \ldots, \tau_k) \to \tau_{k+1},\, \phi \rangle]\!] \tag{5.69}$$

We proceed by fixed-point induction (Manna 1974, Winskel 1993); first we prove

$$\forall n \geq 0 \quad \mathcal{F}^n(\bot) \in \mathcal{T}_\Sigma[\![\langle \forall \vec{\alpha}.\, (\tau_1, \ldots, \tau_k) \to \tau_{k+1},\, \phi \rangle]\!] \tag{5.70}$$

by induction on n. The base case is immediate because $\mathcal{F}^0(\bot) = \bot$ and $\bot$ belongs to the semantics of any functional type. For the step case, observe

$$\begin{aligned}\mathcal{F}^{n+1}(\bot) &\overset{\text{def}}{=} \mathcal{F}(\mathcal{F}^n(\bot)) \\ &= \lambda\langle v_1, \ldots, v_k\rangle.\ \mathcal{E}[\![\, e \,]\!]\ \varphi[f \mapsto \mathcal{F}^n(\bot)]\ \rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]\end{aligned}$$

Using the induction hypothesis $\mathcal{F}^n(\bot) \in \mathcal{T}_\Sigma[\![\langle\forall\vec{\alpha}.\,(\tau_1, \ldots, \tau_k) \to \tau_{k+1},\, \phi\rangle]\!]$ together with Theorem 5.21 applied to hypothesis (5.68), we proceed as in the proof for the [*Fun*] rule and obtain

$$\mathcal{F}^{n+1}(\bot) \in \mathcal{T}_\Sigma[\![\langle\forall\vec{\alpha}.\,(\tau_1, \ldots, \tau_k) \to \tau_{k+1},\, \phi\rangle]\!]$$

This concludes the inductive proof of (5.70). By Lemma 5.16 (inclusiveness of the type semantics), result (5.70) implies (5.69).

This concludes the proof of Theorem 5.23. □

Note that the above proof requires the stronger conclusion provided by Theorem 5.21, i.e. that, given a judgement

$$x_1 : \tau_1, \ldots, x_k : \tau_k,\ \ldots \vdash_{\mathsf{SIZE}} e : \tau_{k+1} \mid \phi\,,$$

the formula ϕ constraints both the type assumptions $x_1 : \tau_1, , \ldots, x_k : \tau_k$ and the result type τ_{k+1}; this is essential to conclude that the abstracted function admits sized type

$$\langle(\tau_1, \ldots, \tau_k) \to \tau_{k+1},\, \phi\rangle$$

as required.

5.5 Size reconstruction algorithm

In Section 5.3 we have defined the size analysis as a proof system deriving sized type judgements for expressions and declarations. In Section 5.4 we proved the correctness of the proof system against the denotational semantics. However, deriving a sized type judgement requires foresight in guessing the correct types and constraints in certain points of the derivation. In order to automate the analysis, we need an algorithm that constructs sized type derivations for un-annotated programs, i.e. a *type reconstruction* algorithm.

5.5.1 Type checking *versus* type reconstruction

The *type checking* problem consists of deciding whether a type derivation is well-formed. Observing the rules in Tables 5.4–5.6 we see that most rules have straightforward syntactical conditions; the only exception is [*Weaken*] which is subject to checking the entailment $\phi \models \psi$ between size constraints. Since entailment for size formulas can be checked algorithmically, e.g. by quantifier elimination (Rabin 1977, Koubarakis 2006), it follows that type checking can be done algorithmically.

The *type reconstruction* problem consists of finding an annotated type and formula τ, ϕ given Γ and e such that such that $\Gamma \vdash_{\mathsf{SIZE}} e : \tau \mid \phi$. This condition expresses the *soundness* requirement for reconstruction, that is, the sized type obtained must be admissible with respect to the proof system. The reconstruction algorithm may also fail, e.g. if no type derivation exists.

A reconstruction algorithm is *complete* if it always computes a "principal" solution, i.e. one that characterises all admissible type derivations. In particular, a complete reconstruction algorithm will fail *only* if no typing derivation exists. The standard example of a complete type reconstruction algorithm is the well-known Damas algorithm W (Damas 1985) that forms the basis of type inference in modern functional programming languages, such as Haskell and ML.

We can formulate principality for the underlying Core Hume type system as follows: let Γ be a closed assumption set[4] and *decl* a function definition; then σ is a *principal type scheme* if and only if:

$$\Gamma \vdash_{\mathsf{DM}} \mathit{decl} : \sigma \tag{5.71}$$

$$\text{for all } \sigma',\ \text{if } \Gamma \vdash_{\mathsf{DM}} \mathit{decl} : \sigma' \text{ then } \sigma \preceq \sigma' \tag{5.72}$$

Condition (5.71) states that σ is a solution for the typing problem; (5.72) states that σ is minimal with respect to the partial order $\preceq$ of instantiation, i.e. *least specific*. For the underlying type system, relation $\preceq$ is defined by

$$\forall \alpha.\, \sigma \preceq [\alpha \mapsto \tau]\, \sigma \tag{5.73}$$

together with rules for reflexivity and transitivity.

Suppose now that we want to extend the notion of principally for sized type schemes: for (sized) type assumptions Γ and a function definition *decl*, the sized type scheme η is *principal* if and only if:

$$\Gamma \vdash_{\mathsf{SIZE}} \mathit{decl} : \eta \tag{5.74}$$

$$\text{for all } \eta',\ \text{if } \Gamma \vdash_{\mathsf{SIZE}} \mathit{decl} : \eta' \text{ then } \eta \preceq \eta' \tag{5.75}$$

For condition (5.75) we must then extend instantiation to *sized* types schemes with the general form $\forall \ell_1 \ldots \ell_n.\, \langle \sigma,\, \phi \rangle$. It is necessary not just to allow instantiations for the size variables ℓ_i but also to allow for replacing the size constraint ϕ by any

[4] For simplicity we state the notion of principal type scheme for *closed* Γ, i.e. such that $\mathrm{FTV}(\Gamma) = \emptyset$.

upper approximation (an application of rule [*Weaken*]). To extend $\preceq$ for sized type schemes we need to allow the possibility of weakening size constraints:

$$\frac{\sigma \preceq \sigma' \qquad \phi \vDash \phi'}{\langle \sigma,\, \phi \rangle \preceq \langle \sigma',\, \phi' \rangle} \tag{5.76}$$

Condition (5.76) implies that the size constraint ϕ in a principal sized type scheme $\forall \ell_1 \ldots \ell_n .\, \langle \sigma,\, \phi \rangle$ must be the *least* with respect to $\vDash$. However, because of the incompleteness of the constraint entailment order, we have already seen that there are functions for which there is no least size constraint (Section 5.2.3). Therefore, our sized type system does not have principal type schemes and consequently we cannot therefore expect a complete reconstruction algorithm.

Dropping the requirement for completeness, we focus on deriving an algorithm that constructs *one* admissible sized type derivation. Although incomplete, the algorithm has two important properties:

1. It always obtains a type derivation when one exists (in fact, when a derivation in the underlying Damas-Milner system exists);

2. In the absence of a "best" size constraint, the algorithm employs a fixed-point iteration method with a *widening operator* to obtain a sound size constraint for recursive functions.[5]

Finally, we shall demonstrate in Chapter 7 that our algorithm yields informative sized types for a representative range of Core Hume programs.

5.5.2 Unification of annotated types

As a preliminary step towards the type reconstruction algorithm, we now look at solving type equality constraints. Since annotated types are terms in a free first-order algebra, we can solve equations using an algorithm based on first-order unification (Robinson 1971).

To simplify the handling of multiple arity type constructors, the input to the unification procedure is a system of type equalities rather than just a pair of types.

Definition 5.24 *A* system of type equality constraints *TE is defined by*

$$TE \ ::= \ \mathsf{true} \ \mid \ \tau = \tau' \wedge TE$$

where τ, τ' are annotated types.

[5] A standard technique from abstract interpretation (Cousot and Cousot 1992*a*,*b*).

We will sometimes write conjunctions of type equalities as a single equation between two sequences of types, i.e. $\vec{\tau} = \vec{\tau}'$ instead of $\tau_1 = \tau_1' \wedge \tau_2 = \tau_2' \wedge \ldots \wedge \tau_n = \tau_n' \wedge \mathsf{true}$.

A solution to a system of type equalities ϵ is a substitution θ on type and size variables that validates all equations; such substitution is a *unifier* of ϵ and we write $\theta \models \epsilon$.

Definition 5.25 *The relation $\theta \models TE$ (read: θ is a unifier of TE) is defined by the rules*

$$\theta \models \mathit{true} \qquad \frac{\theta\,\tau \equiv \theta\,\tau' \qquad \theta \models TE}{\theta \models \tau = \tau' \wedge TE}$$

where $\equiv$ is syntactical equality between types.

Table 5.11 presents the unification algorithm as a procedure $\mathcal{U}$ that, given a system of type equalities TE, either returns a unifier substitution or fails if no such substitution exists. The algorithm processes by replacing, at each step, one equation by simpler set of equivalent equations. This decomposition continues until it reaches either a trivial equality or an equality between a variable and a type; in the latter case, a new binding is recorded in the result substitution and propagated to the pending equations.[6] Unification terminates when all equalities have been eliminated.

The only departure from standard first-order unification occurs when binding an *unsized* type variable; this uses an auxiliary procedure $\mathcal{X}$ that takes a list of types and yields an substitution mapping all annotations to $\boldsymbol{\omega}$ and all variables to unsized. In order to avoid name clashes, unused variables are required at some points, indicated by a side-condition such as "α is new". This could be formalised in the usual manner, e.g. by supplying a list of unused variables to procedures $\mathcal{X}$ and $\mathcal{U}$. We refrain from doing so to avoid cluttering the presentation.

The following lemmas establish the correctness and minimality of unification.

Lemma 5.26 *If $\mathcal{U}(TE) = \theta$ then θ is a* proper substitution*, i.e. θ is idempotent and θ maps unsized type variables to unsized types.*

Proof: These properties are easily verified to be invariants of each of the cases defining $\mathcal{U}$.

Lemma 5.27 *If $\mathcal{U}(TE) = \theta$ then $\theta \models TE$.*

Proof: It is enough to verify that for each of the cases defining $\mathcal{U}$, we have $\theta \models TE$ if and only if $\theta \models TE'$, where TE' is the reduced system on the right-hand side of the definition.

[6] Provided the *occurs check* holds, i.e. variable does not occur in the type; otherwise there is no solution to the type equality.

α, β	sized type variables
$\widehat{\alpha}$, $\widehat{\beta}$	unsized type variables
ℓ, ℓ'	size variables
$\boldsymbol{\omega}$	unbounded size

$$
\begin{array}{rcl}
\mathcal{U}(\mathsf{true}) & = & id \\
\mathcal{U}(\alpha = \beta \ \wedge \ TE) & = & [\alpha \mapsto \beta] \circ \mathcal{U}([\alpha \mapsto \beta]\ TE) \\
\mathcal{U}(\widehat{\alpha} = \widehat{\beta} \ \wedge \ TE) & = & [\widehat{\alpha} \mapsto \widehat{\beta}] \circ \mathcal{U}([\widehat{\alpha} \mapsto \widehat{\beta}]\ TE) \\
\mathcal{U}(\alpha = \widehat{\beta} \ \wedge \ TE) & & \\
= \mathcal{U}(\widehat{\beta} = \alpha \ \wedge \ TE) & = & [\alpha \mapsto \widehat{\beta}] \circ \mathcal{U}([\alpha \mapsto \widehat{\beta}]\ TE) \\
\mathcal{U}(\alpha = \tau \ \wedge \ TE) & = & \begin{cases} [\alpha \mapsto \tau] \circ \mathcal{U}([\alpha \mapsto \tau]\ TE)\,, & \text{if } \alpha \notin \mathrm{FTV}(\tau) \\ \text{fails} & \text{otherwise} \end{cases} \\
\mathcal{U}(\widehat{\alpha} = \tau \ \wedge \ TE) & = & \begin{cases} \text{let } \ \theta = \mathcal{X}(\tau : [\,]) & \\ \qquad \theta' = [\widehat{\alpha} \mapsto \theta\tau] & \\ \ \text{in } \theta' \circ \theta \circ \mathcal{U}(\theta'\theta\ TE)\,, & \text{if } \widehat{\alpha} \notin \mathrm{FTV}(\tau) \\ \text{fails} & \text{otherwise} \end{cases} \\
\mathcal{U}(\mathsf{D}^{\ell}\,\vec{\tau} = \mathsf{D}^{\ell'}\,\vec{\tau}' \ \wedge \ TE) & = & [\ell \mapsto \ell'] \circ \mathcal{U}([\ell \mapsto \ell']\,(\vec{\tau} = \vec{\tau}' \ \wedge \ TE)) \\
\mathcal{U}(\mathsf{D}^{\ell}\,\vec{\tau} = \mathsf{D}^{\boldsymbol{\omega}}\,\vec{\tau}' \ \wedge \ TE) & & \\
= \mathcal{U}(\mathsf{D}^{\boldsymbol{\omega}}\,\vec{\tau}' = \mathsf{D}^{\ell}\,\vec{\tau} \ \wedge \ TE) & = & [\ell \mapsto \boldsymbol{\omega}] \circ \mathcal{U}([\ell \mapsto \boldsymbol{\omega}]\,(\vec{\tau} = \vec{\tau}' \ \wedge \ TE)) \\
\mathcal{U}(\mathsf{D}^{\boldsymbol{\omega}}\,\vec{\tau} = \mathsf{D}^{\boldsymbol{\omega}}\,\vec{\tau}' \ \wedge \ TE) & = & \mathcal{U}(\vec{\tau} = \vec{\tau}' \ \wedge \ TE) \\
\mathcal{U}(\tau = \tau' \ \wedge \ TE) & & \text{fails if } \tau,\ \tau' \text{ have different type constructors}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{X}([\,]) & = & id \\
\mathcal{X}(\alpha : ts) & = & [\alpha \mapsto \widehat{\beta}] \circ \mathcal{X}([\alpha \mapsto \widehat{\beta}]\,ts), \qquad \widehat{\beta} \text{ is new} \\
\mathcal{X}(\widehat{\alpha} : ts) & = & \mathcal{X}(ts) \\
\mathcal{X}(\mathsf{D}^{\ell}\,\vec{\tau} : ts) & = & [\ell \mapsto \boldsymbol{\omega}] \circ \mathcal{X}([\ell \mapsto \boldsymbol{\omega}]\,(\vec{\tau} \mathbin{+\!\!+} ts)) \\
\mathcal{X}(\mathsf{D}^{\boldsymbol{\omega}}\,\vec{\tau} : ts) & = & \mathcal{X}(\vec{\tau} \mathbin{+\!\!+} ts)
\end{array}
$$

Table 5.11: Unification and size erasure of annotated types.

Lemma 5.28 *If there exists θ' such that $\theta' \models TE$, then $\theta = \mathcal{U}(TE)$ is defined and there exists θ'' such that $\theta' = \theta'' \circ \theta$.*

5.5.3 Algorithmic presentation of the typing rules

We present the reconstruction algorithm as a proof system for deriving sized type judgements $\Gamma \vdash_{\textsf{SIZE-R}} e : \tau \mid \phi;\, \theta$ extended with a unifier substitution θ. These extended judgements should be sound with respect to the original proof system in the sense that if $\Gamma \vdash_{\textsf{SIZE-R}} e : \tau \mid \phi;\, \theta$, then $(\theta\,\Gamma) \vdash_{\textsf{SIZE}} e : \tau \mid (\theta\,\phi)$.

However, unlike the proof system of Tables 5.4–5.6, all rules for the extended judgements are syntax-directed, that is, a single rule applies for each syntactical class of expression. This means that we will be able to read the rules for extended judgements as a reconstruction algorithm that takes Γ and e as inputs and yields τ, ϕ and θ as outputs.

Type reconstruction for expressions

To obtain the algorithmic rules from the proof rules of Table 5.4–5.6 we need to replace any implicit type equalities in the antecedents by explicit unifications. Consider the rule for function application:

$$\frac{\Gamma \vdash_{\textsf{INST}} f : \langle \vec{\tau} \to \tau',\, \phi' \rangle \qquad \Gamma \vdash_{\textsf{SIZE}} \vec{e} : \vec{\tau} \mid \phi}{\Gamma \vdash_{\textsf{SIZE}} f\ \vec{e} : \tau' \mid \phi \wedge \phi'}$$

This rule is applicable only if the function's domain type matches the argument type. To change this into an algorithmic rule, we introduce an explicit unification and thread the result substitution:

$$\frac{\Gamma \vdash_{\textsf{INST}} f : \langle \vec{\tau} \to \tau',\, \phi' \rangle \qquad \Gamma \vdash_{\textsf{SIZE-R}} \vec{e} : \vec{\tau}' \mid \phi;\, \theta_1}{\Gamma \vdash_{\textsf{SIZE-R}} f\ \vec{e} : (\theta_2\theta_1\tau') \mid \phi \wedge \phi';\, \theta_2 \circ \theta_1} \quad \theta_2 = \mathcal{U}(\vec{\tau} = \vec{\tau}')$$

Recall that substitutions bind both type and size variables so they must be applied to types *and* constraints. We apply the unifier substitution to the result type eagerly to allow further unifications. However, size constraints are only collected during expression type reconstruction and so we can delay the application of the substitution until a later stage, e.g. just before constraint simplification.

Table 5.12 presents the complete type reconstruction rules for expressions. Note that there are no non-syntax-directed rules like [*Weaken*] and [*Unsize*] in the original type system. Weakening will be used only to simplify constraints at the function declarations level; and $\boldsymbol{\omega}$-substitutions are introduced only by unification and size erasure.

The following lemma states the soundness of type reconstruction.

$$\boxed{\Gamma \vdash_{\textsf{SIZE-R}} e : \tau \mid \phi;\, \theta}$$

$$[\mathit{Int}] \quad \Gamma \vdash_{\textsf{SIZE-R}} n : \mathsf{Int}^{\ell} \mid \ell = n;\ id \qquad (\ell \text{ is new})$$

$$[\mathit{Var}] \quad \frac{\Gamma \vdash_{\textsf{INST-R}} x : \langle \tau,\, \phi \rangle}{\Gamma \vdash_{\textsf{SIZE-R}} x : \tau \mid \phi;\ id}$$

$$[\mathit{FunAp}] \quad \frac{\Gamma \vdash_{\textsf{INST-R}} \Gamma(f) : \langle \vec{\tau} \to \tau'',\, \phi' \rangle \qquad \Gamma \vdash_{\textsf{SIZE-R}} \vec{e} : \vec{\tau}' \mid \phi;\, \theta_1}{\Gamma \vdash_{\textsf{SIZE-R}} f\ \vec{e} : (\theta_2 \theta_1\, \tau'') \mid \phi \wedge \phi';\, \theta_2 \circ \theta_1} \quad \theta_2 = \mathcal{U}(\vec{\tau} = \vec{\tau}')$$

$$[\mathit{ConsAp}] \quad \frac{\Gamma \vdash_{\textsf{INST-R}} \Gamma(c) : \langle \vec{\tau} \to \tau'',\, \phi' \rangle \qquad \Gamma \vdash_{\textsf{SIZE-R}} \vec{e} : \vec{\tau}' \mid \phi;\, \theta_1}{\Gamma \vdash_{\textsf{SIZE-R}} c\ \vec{e} : (\theta_2 \theta_1\, \tau'') \mid \phi \wedge \phi';\, \theta_2 \circ \theta_1} \quad \theta_2 = \mathcal{U}(\vec{\tau} = \vec{\tau}')$$

$$[\mathit{Let}] \quad \frac{\begin{array}{c} \Gamma \vdash_{\textsf{SIZE-R}} e_1 : \tau_1 \mid \phi_1;\, \theta_1 \\ x : \tau_1,\, \theta_1\, \Gamma \vdash_{\textsf{SIZE-R}} e : \tau_2 \mid \phi_2;\, \theta_2 \end{array}}{\Gamma \vdash_{\textsf{SIZE-R}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \mid \phi_1 \wedge \phi_2;\, \theta_2 \circ \theta_1}$$

$$[\mathit{Case}_2] \quad \frac{\begin{array}{l} \Gamma \vdash_{\textsf{SIZE-R}} e_0 : \tau_0' \mid \phi_0;\, \theta_0 \qquad \Gamma \vdash_{\textsf{INST-R}} c_i : \langle \vec{\tau}_i'' \to \tau_i',\, \phi_i' \rangle \quad (i \in \{1,2\}) \\ \vec{x}_1 : \theta_0' \theta_0 \vec{\tau}_1'',\, \theta_0'\, \theta_0\, \Gamma \vdash_{\textsf{SIZE-R}} e_1 : \tau_1 \mid \phi_1;\, \theta_1 \\ \vec{x}_2 : \theta_1 \theta_0' \theta_0 \vec{\tau}_2'',\, \theta_1\, \theta_0'\, \theta_0\, \Gamma \vdash_{\textsf{SIZE-R}} e_2 : \tau_2 \mid \phi_2;\, \theta_2 \end{array}}{\begin{array}{c} \Gamma \vdash_{\textsf{SIZE-R}} \mathsf{case}\ e_0\ \mathsf{of}\ \{\, c_i\ \vec{x}_i \to e_i \,\}_{i \in \{1,2\}} : (\theta_3 \theta_2\, \tau_1) \mid \\ \qquad \phi_0 \wedge \bigvee_{i \in \{1,2\}} (\phi_i \wedge \phi_i');\, \theta_3 \circ \theta_2 \circ \theta_1 \circ \theta_0' \circ \theta_0\,, \\ \text{where } \theta_0' = \mathcal{U}(\tau_0' = \tau_1' \,\wedge\, \tau_0' = \tau_2'),\ \theta_3 = \mathcal{U}(\theta_2\, \tau_1 = \tau_2) \end{array}}$$

$$[\mathit{Tuple}_0] \quad \Gamma \vdash_{\textsf{SIZE-R}} (\,) : () \mid \mathit{True};\ id$$

$$[\mathit{Tuple}_2] \quad \frac{\Gamma \vdash_{\textsf{SIZE-R}} e_1 : \tau_1 \mid \phi_1;\, \theta_1 \qquad \theta_1\, \Gamma \vdash_{\textsf{SIZE-R}} e_2 : \tau_2 \mid \phi_2;\, \theta_2}{\Gamma \vdash_{\textsf{SIZE-R}} (e_1,\, e_2) : (\theta_2 \tau_1, \tau_2) \mid \phi_1 \wedge \phi_2;\, \theta_2 \circ \theta_1}$$

Table 5.12: Algorithmic size typing rules rules for expressions.

$$\boxed{\Gamma \vdash_{\textsf{INST-R}} \eta : \langle \nu,\, \phi \rangle}$$

$$[\mathit{Axiom}] \quad \Gamma \vdash_{\textsf{INST-R}} \langle \nu,\, \phi \rangle : \langle [\vec{\ell} \mapsto \vec{\ell'}]\, \nu,\, \phi \wedge \vec{\ell} = \vec{\ell'} \rangle \qquad \begin{array}{l} \vec{\ell} = \mathrm{FZV}(\nu) \cap \mathrm{FZV}(\Gamma), \\ \vec{\ell'} \text{ are new, } |\vec{\ell'}| = |\vec{\ell}| \end{array}$$

$$[\mathit{Elim}\forall] \quad \frac{\Gamma \vdash_{\textsf{INST-R}} \eta : \langle \nu,\, \phi \rangle}{\Gamma \vdash_{\textsf{INST-R}} \forall \ell .\, \eta : \langle [\ell \mapsto \ell']\, \nu,\, [\ell \mapsto \ell']\, \phi \rangle} \qquad \ell' \text{ is new}$$

$$[\mathit{Elim}\forall\mathit{1}] \quad \frac{\Gamma \vdash_{\textsf{INST-R}} \langle \sigma,\, \phi \rangle : \langle \nu,\, \phi \rangle}{\Gamma \vdash_{\textsf{INST-R}} \langle \forall \alpha .\, \sigma,\, \phi \rangle : \langle [\alpha \mapsto \alpha']\, \nu,\, \phi \rangle} \qquad \alpha' \text{ is new, } \alpha,\, \alpha' \text{ are sized}$$

$$[\mathit{Elim}\forall\mathit{2}] \quad \frac{\Gamma \vdash_{\textsf{INST-R}} \langle \sigma,\, \phi \rangle : \langle \nu,\, \phi \rangle}{\Gamma \vdash_{\textsf{INST-R}} \langle \forall \widehat{\alpha} .\, \sigma,\, \phi \rangle : \langle [\widehat{\alpha} \mapsto \widehat{\alpha}']\, \nu,\, \phi \rangle} \qquad \widehat{\alpha}' \text{ is new, } \widehat{\alpha},\, \widehat{\alpha}' \text{ are unsized}$$

Table 5.13: Algorithmic typing rules for assumption instantiation.

Lemma 5.29 *If* $\Gamma \vdash_{\textsf{SIZE-R}} e : \tau \mid \phi;\, \theta$ *then* $(\theta\, \Gamma) \vdash_{\textsf{SIZE}} e : \tau \mid (\theta\, \phi)$.

Type reconstruction for assumption instantiation

The algorithmic judgements for assumption instantiation have the form $\Gamma \vdash_{\textsf{INST-R}} \eta : \langle \nu,\, \phi \rangle$; the assumptions Γ and size quantified type η are inputs and ν and ϕ are the outputs. Note that since instantiation does not impose any type equalities, unification is not used and hence no substitution is returned.

The algorithmic instantiation rules presented in Table 5.5 follow the structure of the sized type scheme η: [*Elim*∀] eliminates one quantified size variable, substituting it by a fresh one; [*Elim*∀*1*] and [*Elim*∀*2*] eliminate sized and unsized type variables, respectively; finally, [*Axiom*] renames any variables that occur free in the type assumptions (this corresponds to uses of rule [*Rename*] in the original type system). As with type reconstruction for expressions, any substitution by $\boldsymbol{\omega}$, if required, will be obtained at a subsequent stage as a result of unification.

The following lemma states the soundness of type reconstruction for assumption instantiation.

Lemma 5.30 *If* $\Gamma(x) = \eta$ *and* $\Gamma \vdash_{\textsf{INST-R}} \eta : \langle \nu,\, \phi \rangle$, *then* $\Gamma \vdash_{\textsf{INST}} x : \langle \nu,\, \phi \rangle$ *and* $\mathrm{FZV}(\Gamma) \cap \mathrm{FZV}(\nu) = \emptyset$.

$$[Abs] \quad \frac{\begin{array}{r} x_1 : \alpha_1, \ldots, x_k : \alpha_k, \Gamma \vdash_{\mathsf{SIZE\text{-}R}} e : \tau \mid \phi; \theta \\ \vec{\ell} = \mathrm{FZV}(\theta\phi) \setminus \mathrm{FZV}((\theta\alpha_1, \ldots, \theta\alpha_k) \to \tau) \\ \phi' = \mathsf{SIMPLIFY}(\exists \vec{\ell}.\, (\theta\phi)) \end{array}}{\Gamma \vdash_{\mathsf{ABS\text{-}R}} \lambda(x_1, \ldots, x_k).\, e : \langle (\theta\alpha_1, \ldots, \theta\alpha_k) \to \tau,\, \phi' \rangle} \quad \alpha_i \text{ are new}$$

$$[Fun] \quad \frac{\Gamma \vdash_{\mathsf{ABS\text{-}R}} \lambda\vec{x}.\, e : \langle \vec{\tau} \to \tau',\, \phi \rangle}{\Gamma \vdash_{\mathsf{SIZE\text{-}R}} \mathsf{let}\ f\ \vec{x} = e : \forall \vec{\ell}.\, \langle \forall \vec{\alpha}.\, \vec{\tau} \to \tau',\, \phi \rangle} \quad \begin{array}{l} \vec{\alpha} = \mathrm{FTV}(\vec{\tau} \to \tau') \setminus \mathrm{FZV}(\Gamma) \\ \vec{\ell} = \mathrm{FZV}(\vec{\tau} \to \tau') \setminus \mathrm{FTV}(\Gamma) \end{array}$$

$$[Rec] \quad \frac{\phi = \mathsf{FIX}(\Gamma, f\ \vec{x} = e : \vec{\tau} \to \tau')}{\Gamma \vdash_{\mathsf{SIZE\text{-}R}} \mathsf{letrec}\ f\ \vec{x} = e : \forall \vec{\ell}.\, \langle \forall \vec{\alpha}.\, \vec{\tau} \to \tau',\, \phi \rangle} \quad \begin{array}{l} \vec{\alpha} = \mathrm{FTV}(\vec{\tau} \to \tau') \setminus \mathrm{FZV}(\Gamma) \\ \vec{\ell} = \mathrm{FZV}(\vec{\tau} \to \tau') \setminus \mathrm{FTV}(\Gamma) \end{array}$$

Table 5.14: Algorithmic typing judgements for function declarations.

```
**Algorithm** FIX: iterate to obtain a size constraint for letrec $f\ \vec{x} = e$
**Inputs:** $\Gamma$, $f\ \vec{x} = e$, $\vec{\tau} \to \tau'$
**Output:** $\phi$
$\vec{\ell} = \mathrm{FZV}(\vec{\tau} \to \tau')$
$i \leftarrow 0$
$\phi \leftarrow \mathit{False}$
loop
    compute $\langle \vec{\tau}' \to \tau'', \phi' \rangle$ using $f : \forall \vec{\ell}.\, \langle \vec{\tau} \to \tau', \phi \rangle, \Gamma \vdash_{\mathsf{ABS\text{-}R}} \lambda\vec{x}.\, e : \langle \vec{\tau}' \to \tau'', \phi' \rangle$
    $\theta = \mathcal{U}(\vec{\tau} = \vec{\tau}' \wedge \tau' = \tau'')$
    assert $\theta\vec{\tau} \equiv \vec{\tau} \wedge \theta\tau' \equiv \tau'$
    $\phi' \leftarrow \mathsf{HULL}(\theta\phi')$
    if $\phi' \models \phi$ then
        return $\phi$
    else
        $\phi \leftarrow \phi \nabla_i \phi'$
    end if
    $i \leftarrow 1 + i$
end loop
```

Table 5.15: Size fixpoint iteration for recursive functions.

Type reconstruction for function declarations

The sized type reconstruction for function declarations is presented in Table 5.14. Both recursive and non-recursive functions use an auxiliary judgement $\Gamma \vdash_{\textsf{ABS-R}} \lambda\vec{x}.\, e : \langle \vec{\tau} \rightarrow \tau',\, \phi\rangle$ to reconstruct the annotated type $\vec{\tau} \rightarrow \tau'$ and size constraint ϕ for a function with formal parameters $\vec{x}$ and body e.[7] As in Damas-Milner type inference, the abstraction judgement reconstructs the type for the body under generic assumptions for the arguments; the inferred types are obtained by applying the unifier substitution to the generic assumptions.

Rule $[\mathit{Abs}]$ invokes a procedure SIMPLIFY to perform constraint simplification; this is discussed in Section 5.5.4. As in other type systems that extend polymorphic types with constraints (Mitchell 1984, Fuh and Mishra 1988, 1989), we simplify constraints before generalisation; this reduces the number of constraints that will be duplicated by type scheme instantiation.

Rules $[\mathit{Fun}]$ and $[\mathit{Rec}]$ perform sized type reconstruction for polymorphic function declarations. In both rules the result type scheme is quantified in all size and type variables that occur in the type but not the assumptions.

To simplify the presentation, rule $[\mathit{Rec}]$ for recursive functions assumes that the annotated type is known in advance. This is done without loss of generality, since the annotated type can be obtained by performing Damas-Milner type inference and annotating all data type constructors with distinct size variables.

The auxiliary procedure FIX constructs a sound size constraint for a recursive function iteratively. The initial approximation is *False* (the bottom element in the lattice of constraints). Each iteration computes the next approximation ϕ' using the previous approximation ϕ in the type assumption for the recursive function. As in the proof system, we quantify the assumption for typing the function body over size variables, i.e. we allow polymorphic recursion on sizes (but not types).

The assertion $\theta\vec{\tau} \equiv \vec{\tau} \wedge \theta\tau' \equiv \tau'$ is required to ensure that the size variables in the annotated type are invariant throughout the fixed point computation. This can be guaranteed by imposing a total order on variables and implementing unification so that substitutions binding two variables respect the ordering.

The loop terminates when $\phi' \vDash \phi$; under that condition, by Lemma 5.29 followed by an application of rules $[\mathit{Weaken}]$ and $[\mathit{Rec}]$, we conclude that ϕ is admissible for f. Since this is the only successful termination condition, we conclude that, when FIX terminates successfully, it yields an admissible size constraint.

[7] We recall that our language is first-order and therefore does not allow nameless functions; the term $\lambda\vec{x}.\, e$ is *not* an expression and we use it only as a part of an abstraction judgement $\cdot \vdash_{\textsf{ABS-R}} \lambda\vec{x}.\, e : \langle\cdot,\, \cdot\rangle$.

To ensure the termination of FIX we employ a widening operator for systems of linear inequalities (Cousot and Halbwachs 1978, Bagnara, Hill, Ricci and Zaffanella 2003) that guarantees the iteration will stabilise in a finite number of steps. More generally, widening operators are a standard technique used in abstract interpretation to accelerate or guarantee convergence of iterations on lattices with large or infinite ascending chains (see Section 2.3.6).

We allow some further flexibility by parameterising the widening on the iteration i; this can be used to delay the use of widening by defining an extrapolation threshold $k > 0$ and choosing

$$\nabla_i \overset{\text{def}}{=} \begin{cases} \uplus\,, & \text{if } i \leq k \\ \nabla & \text{otherwise} \end{cases}$$

where $\uplus$ is the convex hull operator and ∇ is the proper widening operator (Halbwachs 1979, Bagnara, Hill, Ricci and Zaffanella 2003). Therefore, FIX uses the more precise upper-bound operator (the convex hull) for the first k iterations before resorting to the widening operator that ensures convergence but potentially loses precision; this is a standard technique in abstract interpretation for improving the precision of fixed point approximations (Cousot and Cousot 1992*b*).

The following lemma states the soundness of declaration reconstruction algorithm.

Lemma 5.31 *If* $\Gamma \vdash_{\textsf{SIZE-R}} decl : \eta$ *then* $\Gamma \vdash_{\textsf{SIZE}} decl : \eta$.

Example 5.32 We apply the sized type reconstruction algorithm to the list append function:

$$\begin{aligned} \mathsf{letrec}\ app\ (xs, ys) = \mathsf{case}\ xs\ \mathsf{of}& \\ &\mathsf{Nil} \to ys \\ &\mathsf{Cons}(x, xs') \to \mathsf{Cons}(x, app(xs', ys)) \\ &: (\mathsf{List}^i\ a, \mathsf{List}^j\ a) \to \mathsf{List}^k\ a \end{aligned} \tag{5.77}$$

The application of FIX to (5.77) constructs the following iteration:

$$\begin{aligned} &\phi_0 \equiv \mathit{False} && \phi'_0 \equiv j = k \wedge i = 0 \\ &\phi_1 \equiv \phi_0 \nabla \phi'_0 \equiv j = k \wedge i = 0 && \phi'_1 \equiv i + j = k \wedge i \leq 1 \wedge 0 \leq i \\ &\phi_2 \equiv \phi_1 \nabla \phi'_1 \equiv i + j = k \wedge 0 \leq i && \phi'_2 \equiv i + j = k \wedge 0 \leq i \end{aligned}$$

We now verify that $\phi'_2 \vDash \phi_2$ and therefore FIX terminates. The inferred sized type for append is $app : \forall ijk.\, \langle \forall a.\, (\mathsf{List}^i\ a, \mathsf{List}^j\ a) \to \mathsf{List}^k\ , i + j = k \wedge 0 \leq i \rangle$. □

5.5.4 Simplifying size constraints

Our type reconstruction judgements for function declarations require auxiliary procedures that manipulate size constraints. We need algorithms not just to decide constraint entailment but also to simplify constraints. Simplification is important not just for efficiency of the reconstruction algorithm (by reducing the size of constraints), but also to make the output of the analysis intelligible. We employ the standard method of performing quantifier elimination (Rabin 1977, Koubarakis 2006) by variable elimination in systems of linear inequations (Schrijver 1986, Chandru 1993) (see also Section 2.3.6).

Elimination of existential quantifiers

Procedure SIMPLIFY eliminates existential quantifiers from a size formula. The method can be applied in general to first-order theory with variable elimination (Rabin 1977); our presentation follows (Koubarakis 2006). To eliminate existential quantifiers from a formula ψ:

1) obtain a formula $\exists\ell_1. \ldots \exists\ell_m. \phi$ equivalent to ψ and in *prenex* form using Lemma 5.5;

2) use the distributivity of $\wedge$ over $\vee$ to rewrite ϕ in disjunctive form, i.e. $\phi \simeq \phi_1 \vee \ldots \vee \phi_n$, where ϕ_i are conjunctions of inequalities;

3) eliminate variables $\ell_1, \ldots, \ell_m$ one at time using variable elimination in systems of linear inequalities,

$$\exists\ell. (\phi_1 \vee \ldots \vee \phi_n) \simeq \mathsf{ELIM}(\ell, \phi_1) \vee \ldots \vee \mathsf{ELIM}(\ell, \phi_n)$$

where ELIM is the procedure for variable elimination in a system of linear inequalities.

The result constraint is a disjunction of conjunctions of linear inequalities. However, it can still include redundant terms; we therefore add a final step to simplify the result:[8]

4) let $\phi_1 \vee \ldots \vee \phi_n$ be the quantifier-free constraint in disjunctive form; remove all ϕ_i such that $\phi_i \vDash \mathit{False}$ or $\phi_i \vDash \phi_j$ for $i \neq j$.

Note that step 4 above requires testing entailment between system of convex linear inequalities rather than general constraints; therefore we use a simpler algorithm for testing containment of convex polyhedra (Schrijver 1986).

[8] This is an adaptation of the normalisation step described in the "powerset construction" of Bagnara, Hill and Zaffanella (2003).

Finally, we remark that although step 1 would be necessary in the general case, it is not required in our reconstruction algorithm because the constraint is always in prenex form (the only algorithmic rule that introduces existential quantifiers is $[Abs]$).

Computation of the convex hull

For each iteration in FIX we use a procedure HULL to compute the *convex hull* of the size constraint.[9] This ensures that iterates are systems of linear inequalities and therefore we can employ a widening operator to ensure convergence.

Employing the convex hull on the constraint of a function reduces the constraint size by eliminating disjunctions (this is particularly effective for functions with many alternative computation paths); the drawback is a potential loss of precision. Our experiments suggested that the situations where this loss is harmful are few and rather specialised. In our implementation we therefore chose to employ the convex hull by default even for non-recursive functions but allow the user to specify otherwise to retain higher precision when required.

Implementation considerations

We have implemented the sized type reconstruction algorithm in Haskell using the Glasgow Haskell Compiler version 6.4.1 (GHC). Note that it is not possible to decouple type reconstruction from size constraint solving, e.g. generate a set of constraints for solving off-line, because entailment checking, hulling and widening are required during fixpoint approximation. Therefore we choose to use a constraint library rather than an external constraint solver, namely the *Parma Polyhedra Library* version 0.9.1 (Bagnara et al. 2006). This library suits our necessities in the following aspects:

1. it specifically targets the requirements of program analyses, e.g. by providing implementations of several widening operators;

2. it emphasises correctness, e.g. by using arbitrary-precision rational arithmetic to avoid floating-point rounding;

3. it imposes no arbitrary limits for data (other than available memory);

4. it is rather efficient and the developers aim at making it even more so;

[9] The convex hull of ϕ is the smallest convex constraint that is an upper-bound of ϕ with respect to $\models$ (see Section 2.3.6).

5. it includes interfaces for C/C++ and other high-level languages[10];
6. it is well documented and actively maintained;
7. it is distributed freely in source-form under the GNU Public Licence.

A web interface to our prototype implementation, including several examples, is available at `http://www.ncc.up.pt/~pbv/cgi/cost.cgi`. This implementation performs not just the size analysis but also the cost analysis for stack and heap that will be presented in Chapter 6.

5.6 Discussion

5.6.1 Partiality

Our sized type system infers a *safety* property, namely an approximation of the sizes of results of successful evaluations; the soundness result (Theorem 5.21) holds trivially for a divergent expression, i.e. that evaluates to bottom. This is unlike the system of Hughes et al. (1996) which verifies the *liveness* properties of termination and productivity: a sized type derivation in their system guarantees that an expression does not evaluate to bottom.

Our analysis can, in many circumstances, accurately determine the domains of functions. For example, consider the erroneous list length function of Section 3.3:

$$\mathit{wronglen}\ xs = \mathsf{case}\ xs\ \mathsf{of}\ \mathsf{Nil} \to 0\ \ |\ \ \mathsf{Cons}\ x\ xs'\ \to\ 1 + \mathit{wronglen}\ xs$$

This function is rejected by the sized type system of Hughes et al. because it diverges for non-empty lists (the recursive call is on xs rather than xs'). By contrast, our size analysis infers a size relation for the limited domain where the *wronglen* is defined.

```
wronglen :: [a]^z1->Int^z2 | z2=0, z1=0
```

The inferred constraint also implies that *wronglen* is undefined for non-empty lists; thus the following application yields an unsatisfiable constraint:

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{INST}} \mathit{wronglen} : \langle \mathsf{List}^i\, a \to \mathsf{Int}^j,\ i = 0 \wedge j = 0\rangle \\ \Gamma \vdash_{\mathsf{SIZE}} xs : \mathsf{List}^i\, a \mid i = 1\end{array}}{\Gamma \vdash_{\mathsf{SIZE}} \mathit{wronglen}\ xs : \mathsf{Int}^j \mid \underbrace{i = 0 \wedge j = 0 \wedge i = 1}_{\mathit{False}}}\ \text{by } [\mathit{FunApp}]$$

[10] Regrettably, the interface to Haskell/GHC is not yet incorporated in the standard distribution PPL version 0.9.1. The author would like to express his thanks to Axel Simon for his help in facilitating access to his experimental interface source code.

Note that this typing is sound because the application evaluates to bottom. Of course one can argue that programmers do not intentionally execute divergent computations, so these situations should be identified as errors.

Our analysis can easily be extended to detect such errors by requiring that each function is annotated with a constraint asserting the expected domain. For our example, this assertion would be:

$$\mathit{wronglen} : \mathsf{List}^i\, a \rightarrow \mathsf{Int}\,, \quad \textbf{domain}\ i \geq 0$$

To allow functions that are partial on the data types, e.g. list head or tail, we assume the domain assertions are provided by the programmer. Thus, the above constraint $i \geq 0$ expresses the programmer's intention that *wronglen* ought to be defined for all lists. Size analysis then proceeds as before, ignoring the assertion, and yields:

$$\mathit{wronglen} : \langle \mathsf{List}^i\, a \rightarrow \mathsf{Int}^j,\ i = 0 \wedge j = 0 \rangle$$

Next the analysis tests if the inferred constraint contains the domain constraint by existentially quantifying the result variables:

$$i \geq 0 \overset{?}{\vDash} \exists j.\, (i = 0 \wedge j = 0) \tag{5.78}$$

But $\exists j.\, (i = 0 \wedge j = 0) \simeq i = 0$ and $i \geq 0 \nvDash i = 0$, so entailment check (5.78) fails and the function is rejected.

Note that there is no guarantee of totality when the domain check succeeds: our analysis proves a safety property, so the right hand side of (5.78) is, in general, over-approximated. However, if the check fails then the function must be less defined than the asserted domain. Thus, the guarantee is a dual to that of the type system of Hughes et al.: our analysis rejects *some* non-terminating programs but *never* rejects terminating ones.

Our analysis accepts non-primitive recursive functions: consider the Ackermann function, a standard example of a total function on naturals that is not primitive recursive:

$$\begin{aligned} \mathit{ack}\ m\ n = {} & \mathsf{if}\ m = 0\ \mathsf{then}\ n + 1 \\ & \mathsf{else\ if}\ n = 0\ \mathsf{then}\ \mathit{ack}\ (m-1)\ 1 \\ & \mathsf{else}\ \mathit{ack}\ (m-1)\ (\mathit{ack}\ m\ (n-1)) \end{aligned} \tag{5.79}$$

This standard first-order definition cannot be typed in the system of Hughes et al. because there is no single decreasing size to allow using the recursion type rule (3.4) (page 55); it can, however, be typed by transforming it into a higher-order primitive-recursive form.

By contrast, our analysis accepts the first-order Ackermann, obtaining the following size information:

```
ack :: {Int^z0,Int^z1}->Int^z2 | z2>=1+z0+z1, 3*z0+2*z2>=2+2*z1
```

As would be expected, only lower bounds are obtained (since Ackermann grows faster than any primitive recursive function):

$$\mathit{ack}\ m\ n \geq 1 + m + n \ \wedge\ \mathit{ack}\ m\ n \geq 1 + n - \frac{3}{2}m\,, \quad \forall m \forall n$$

The size constraint expresses no restriction of the function domain; this is to be expected since Ackermann is a total function on the naturals.[11]

5.6.2 Rational size relations

The algorithm of Section 5.5.4 solves the size constraints over rationals rather than integers; this not only simplifies elimination of quantifiers but also allows for a more compositional combination of size bounds. To exemplify this, consider two recursive lists functions: *foo* deletes every third element from a list; *bar* that inserts an element for every two elements; and *foobar* is a composition of the two functions.[12]

```
foo :: [a] -> [a]
foo (x1:x2:x3:xs) = x1:x2:foo xs ;
foo xs = xs ;

bar :: [a] -> [a]
bar (x1:x2:xs) = x1:x2:x1:bar xs ;
bar xs = xs ;

foobar :: [a] -> [a]
foobar xs = bar (foo xs);
```

Note that the second equations of *foo* and *bar* apply only if the first does not, i.e. if the argument list has too few elements.

We can guess "asymptotic" size relations by simple inspection of the equations: *foo* consumes three elements for each two produced; and *bar* consumes two for each three produced; therefore we have

$$|\mathit{foo}\ xs| \approx \frac{2}{3}|xs| \qquad |\mathit{bar}\ xs| \approx \frac{3}{2}|xs|\,,$$

[11] Note that (5.79) is also defined for some *negative* arguments so that it is *not* true that $m \geq 0$ and $n \geq 0$.

[12] This examples are due to Hofmann and Jost (2003). For legibility we use pattern-matching equations; the translation into our core language is standard.

where $|\cdot|$ is the list length; composing the size relations for the two functions yields:

$$|foobar\ xs| \approx |bar\ (foo\ xs)| \approx \frac{3}{2} \times \frac{2}{3} \times |xs| \approx |xs|\ .$$

However, if we are interested (for example) in bounding the heap space for *foobar*, we need a more precise bound on $|foobar\ xs|$ in terms of $|xs|$. Let us start by considering *foo*; we analyse separately three cases corresponding to the remainders of $|xs|$ by 3:

$$|foo\ xs| = \begin{cases} 2k & \text{if } |xs| = 3k \\ 2k+1 & \text{if } |xs| = 3k+1 \\ 2k+2 & \text{if } |xs| = 3k+2 \end{cases}$$

A similar analysis for *bar* would yield *two* more cases (for the remainders by 2), leading to *six* cases for *foobar*; these could be simplified into:

$$|foobar\ xs| = \begin{cases} 1+|xs| & \text{if } |xs| = 3k+2 \\ |xs| & \text{otherwise} \end{cases} \tag{5.80}$$

Although this kind of analysis could in principle be automated, e.g. using the Omega calculator (Pugh 1992), the number of alternatives and consequently, the number of constraints, can grow exponentially with the number of applications. Moreover, the case analysis on the remainders is unnecessary when we are interested only in the lower and upper bounds.

Our alternative is to obtain solutions for rationals instead of integers. A first advantage is that there is no need to consider congruence relations for quantifier elimination. The second advantage is that we gain the possibility of employing many approximation techniques of convex linear inequalities to trade precision for efficiency, e.g. widening and hulling; see Section 5.5.

For the above definitions of *foo*, *bar* and *foobar*, our analysis yields the following sized types (where all simplifications were performed automatically):

```
foo :: [a]^z1->[a]^z2 | 3*z2>=2*z1, 2+2*z1>=3*z2, z1>=z2
bar :: [a]^z1->[a]^z2 | 1+2*z2>=3*z1, z2>=z1, 3*z1>=2*z2
foobar :: [a]^z1->[a]^z2 | 1+2*z2>=2*z1, 3*z2>=2*z1, 3*z1>=2*z2,
   1+z1>=z2
```

Note that although the list sizes are integers, the constraints represent potentially rational solutions. This means that we should interpret the results by isolating variables and taking the integral approximations of the left and right bounds. For the result sizes of *foo* and *bar* this yields:

$$\left\lceil \frac{2}{3}|xs| \right\rceil \leq |\mathit{foo}\ xs| \leq \left\lfloor \frac{2}{3} + \frac{2}{3}|xs| \right\rfloor$$

$$|\mathit{foo}\ xs| \leq |xs|$$

$$\left\lceil \frac{3}{2}|xs| - \frac{1}{2} \right\rceil \leq |\mathit{bar}\ xs| \leq \left\lfloor \frac{3}{2}|xs| \right\rfloor$$

$$|xs| \leq |\mathit{bar}\ xs|$$

Finally, for the size of *foobar* this yields:

$$\left\lceil |xs| - \frac{1}{2} \right\rceil \leq |\mathit{foobar}\ xs| \leq 1 + |xs|$$

$$\Longleftrightarrow |xs| \leq |\mathit{foobar}\ xs| \leq 1 + |xs|$$

because $\lceil |xs| - 1/2 \rceil = |xs|$. We get the same upper and lower bounds on the length of *foobar* of (5.80); these are tight bounds, as can be witnessed by

$$\begin{aligned} \mathit{foobar}\ [1] &= \mathit{bar}\ (\mathit{foo}\ [1]) & \qquad \mathit{foobar}\ [1,\ 2] &= \mathit{bar}\ (\mathit{foo}\ [1,\ 2]) \\ &= \mathit{bar}\ [1] & &= \mathit{bar}\ [1,\ 2] \\ &= [1] & &= [1,\ 2,\ 1] \end{aligned}$$

5.6.3 Limitations regarding collection types

Our size analysis suffers from the same limitation as the system of Chin and Khoo (2001): while we can infer the sizes of collections, e.g. lists, trees or vectors, we do not infer sizes of values *inside* collections. In fact, allowing size annotations inside collection types (other than $\boldsymbol{\omega}$) can lead to *unsound* sized type derivations.

To exemplify the problem, consider sized type assumptions Σ for constructors of (monomorphic) lists of integers that express both the list length and the sizes of inner values:

$$\begin{aligned} \Sigma \overset{\text{def}}{=} \mathsf{Nil} &: \forall ik.\, \langle () \to \mathsf{List}^i\, \mathsf{Int}^k,\ i = 0 \rangle, \\ \mathsf{Cons} &: \forall ijk.\, \langle (\mathsf{Int}^k, \mathsf{List}^i\, \mathsf{Int}^k) \to \mathsf{List}^j\, \mathsf{Int}^k,\ j = 1 + i \wedge i \geq 0 \rangle \end{aligned} \tag{5.81}$$

Now consider an expression that calculates the difference between the first and second integers in a list xs,

$$\begin{aligned} &\mathsf{case}\ xs\ \mathsf{of} \\ &\qquad \mathsf{Cons}\ (x_1, xs') \to \mathsf{case}\ xs'\ \mathsf{of} \\ &\qquad\qquad \mathsf{Cons}\ (x_2, xs'') \to \mathsf{sub}\ (x_1, x_2) \end{aligned} \tag{5.82}$$

where sub is a primitive operation with sized type:

$$\mathsf{sub} : \forall ijk.\, \langle (\mathsf{Int}^i, \mathsf{Int}^j) \to \mathsf{Int}^k,\ k = i - j \rangle$$

Under an assumption that xs is a list of integers, it is straightforward to obtain a typing for (5.82) in our type system. For brevity, we omit the complete derivation and present just the expression annotated with the intermediate types and size constraints:

$$\begin{array}{l}
\textsf{case}\ \ \overbrace{xs}^{\textsf{List}^n\ \textsf{Int}^k}\ \ \textsf{of}\\
\quad \textsf{Cons}\ (\underbrace{x_1}_{\textsf{Int}^k},\ \ \underbrace{xs'}_{\textsf{List}^{n_1}\ \textsf{Int}^k}\ ) \rightarrow \textsf{case}\ xs'\ \textsf{of}\\
\qquad \textsf{Cons}\ (\underbrace{x_2}_{\textsf{Int}^k},\ \ \underbrace{xs''}_{\textsf{List}^{n_2}\ \textsf{Int}^k}\ ) \rightarrow \underbrace{\textsf{sub}\ (x_1, x_2)}_{\textsf{Int}^r}\\
\text{subject to:}\quad n = 1 + n_1 \wedge n_1 \geq 0 \wedge n_1 = 1 + n_2 \wedge n_2 \geq 0 \wedge r = k - k
\end{array} \tag{5.83}$$

From (5.82) we can derive the type judgement

$$xs : \textsf{List}^n\ \textsf{Int}^k,\ \Sigma \vdash_{\textsf{SIZE}} (5.82) : \textsf{Int}^r \mid n \geq 2 \wedge r = 0$$

where we have simplified the result constraint by eliminating variables n_1 and n_2 that do not occur in the result type or in the assumptions. Although the size constraint correctly captures the minimal length of xs, the size $r = 0$ of result is *unsound* because the first and second list elements can differ.

The unsoundness is caused by insufficient size polymorphism in the Cons assumption of (5.81): the annotation of the type $\textsf{Int}^k$ of the list elements constrains *every* element to have the same size k. This can be be exemplified by calculating the size constraint $\mathcal{S}$ defined in Section 5.4 of a list with two distinct integers:

$$\begin{array}{l}
\mathcal{S}_\Sigma(\lfloor\langle\textsf{Cons},\ \langle 0,\ \langle\textsf{Cons},\ \langle 1,\ \langle\textsf{Nil},\ \mathbf{u}\rangle\rangle\rangle\rangle\rangle\rfloor :: \textsf{List}^n\ \textsf{Int}^k)\\
\stackrel{\text{def}}{=} \mathcal{S}_\Sigma(\lfloor 0\rfloor :: \textsf{Int}^k) \wedge \mathcal{S}_\Sigma(\lfloor 1\rfloor :: \textsf{Int}^k) \wedge n = 1 + n_1 \wedge n_1 = 1 + n_2 \wedge n_2 = 0\\
\simeq k = 0 \wedge k = 1 \wedge \ldots \simeq \mathit{False}
\end{array}$$

The example demonstrates that the assumptions (5.81) are not *constructor consistent* (Definition 5.9 of Section 5.4): the list denotation is a non-bottom value that has an unsatisfiable size. This invalidates one of the preconditions for our soundness result (Theorem 5.21) and indeed we derive an unsound size information.

Our inference algorithm presented in Section 5.5 ensures constructor consistency by removing all size information for types inside collections, i.e. substituting all size annotations by $\boldsymbol{\omega}$ and all type variables by unsized ones.

While this is an important quality limitation, we remark that it will not prevent obtaining cost bounds when these depend only on the sizes an outermost data structure. Furthermore, algorithms that depend on sizes of nested data are also likely to have non-linear costs which would not be expressible using Presburger constraints. We therefore do not address this limitation in this thesis and leave it as a subject for future research.

Chapter 6

Cost analysis

In this chapter we extend the sized type system of Chapter 5 to perform static cost analysis for Hume expressions. As a proof-of-concept, our analysis will model the dynamic memory requirements of a prototype abstract machine.

The development is as follows: in Section 6.2 we define an operational semantics for expressions as an abstract machine; in Section 6.3 we define an operational semantics for the coordination layer as a transition system. In Section 6.4 we extend the size analysis of Chapter 5 with *cost annotations* and *cost effects* modelling the stack and heap usage of our abstract machine. In Section 6.5 we fomulate the soundness of the cost analysis with respect to a denotational semantics instrumented with costs. We present extensions for common space optimisations in Section 6.6 and show how to extend the analysis for the coordination layer in Section 6.7.

6.1 Overview

We start from the observation that the notion of "cost" is inherently operational: it is a property of the finite nature of computation rather than of the denoted values. Therefore, we must consider an *operational* model of computation in order to reason about costs.

Since we are interested in modelling realistic *stack* and *heap* space costs, we will consider in Section 6.2 a "small-step" rather than "big-step" semantics for expression evaluation. In Section 6.3 we extend the operational semantics to model space re-use at the coordination layer by presenting a *scheduling semantics* where communication wires are implemented as bounded memory regions. By predicting safe bounds for these regions and for dynamic stack, our cost analysis will obtain guarantees of bounded space behaviour for a complete network of Core Hume boxes.

6.2 Operational semantics for expressions

In this section we define an operational semantics for Core Hume expressions in the form of state transitions of an abstract machine. Instead of a "big-step" semantics, we start from a "small-step" one that explicitly records control-flow in a stack to properly account stack costs. Our abstract machine is based on the SECD machine (Landin 1964, Kogge 1991).

Our operational semantics interprets expressions rather than a compiled instruction stream. Thus, in the classification of Ager et al. (2003*a*), it is an *abstract* rather than a *virtual* machine. To avoid the need to perform substitutions, we use an environment that binds values to free identifiers in expressions. It would be straightforward to compile Core Hume expressions into an instruction set, replacing reference to identifiers by stack offsets. Such translation does not influence stack and heap usage and therefore we will not pursue it here.

The machine configuration consists of a *control*, an *environment*, a *stack* and a *heap*. Each of these components represents part of the evaluation context:

the Control is a pseudo-instruction stream specifying the pending evaluations;

the Environment is an association of free identifiers to values;

the Stack is the storage area for temporary values and continuations;

the Heap is the storage area for structured values (e.g. tuples, lists, etc).

Following Hughes and Pareto (1999), we have coalesced the separate dump and value stacks of the original SECD machine into a single stack; this could ultimately be implemented as the system stack of a general purpose or embedded computer and assures that our stack cost model mimics a realistic implementation.

6.2.1 Region-based memory management

Our abstract machine employs a simple region-based memory management strategy, where the heap is split into a number of smaller sections called *regions* (Tofte and Talpin 1997).

A heap value is referenced by an *address*: a pair (r, o) of a *region identifier* r and an *offset* o within the region.

$$a \in \mathbf{Addr} \overset{\text{def}}{=} \mathbf{Region} \times \mathbf{Offset}$$

Heap values can be allocated in any region and, therefore, regions grow independently of each other. However, individual values in regions are not deallocated; instead, a whole region is *reset* when its values are no longer needed, i.e. all its values are deallocated (but the region itself remains) (Tofte et al. 2004).

Unlike the system of Tofte and Talpin, regions in Core Hume are not dynamic: the number of regions is determined statically by the network of communication wires. Dynamically, we require only two operations on regions:

1. allocate a new value in a region;
2. reset a region (i.e. deallocate all values in the region).

Boxed and unboxed values

We distinguish two kinds of values: *boxed values* that require a runtime tag and *unboxed values* that do not.

$$
\begin{array}{lcll}
\text{Unboxed values} \quad u & \in & \textbf{Unboxed} & \\
u & ::= & n & \text{primitive integer} \\
 & | & a & \text{heap address} \\
\text{Boxed values} \quad b & \in & \textbf{Boxed} & \\
b & ::= & \langle c, u_1, \ldots, u_n \rangle & \text{tagged tuple } (n \geq 0)
\end{array}
$$

An unboxed value is either a primitive integer or a heap address. A boxed value is a tagged sequence $\langle c, u_1, \ldots, u_n \rangle$ of unboxed values; the tag c ranges over a finite set of constructors (i.e. those explicitly mentioned in the program). Each of the fields u_i is an unboxed value (i.e. either a primitive integer or an address). The number n of fields in a boxed value can be zero, e.g. for the boolean values $\langle \mathsf{True} \rangle$ and $\langle \mathsf{False} \rangle$ or the empty list constructor $\langle \mathsf{Nil} \rangle$.

Unboxed values occupy a fixed and small amount of storage space (e.g. one machine word), while boxed values occupy variable storage space. Therefore, unboxed values are allocated in the stack, and boxed values are allocated in the heap and referenced by an address.

Allocation in a region

The *heap* is a finite map from addresses to boxed values:

$$H \in \textbf{Heap} \overset{\text{def}}{=} \textbf{Addr} \to_{\text{fin}} \textbf{Boxed}$$

Heap allocation is always done in a specific region. To ensure that space associated with "old" regions can be re-used, we will not allow cross-region references in the heap. To formalise this, we introduce a relation 'u **at** r' meaning that an unboxed value u is *compatible* with region r:

$$\begin{aligned}(r', o) \ \textbf{at}\ r &\overset{\text{def}}{\Longleftrightarrow} r = r' \\ n\ \textbf{at}\ r &\overset{\text{def}}{\Longleftrightarrow} \text{True}\end{aligned}$$

An address is only compatible with its associated region while an integer is compatible with any region. Thus, when allocating a new boxed value $\langle c, u_1, \ldots, u_n\rangle$ in region r we will impose the side-conditions u_i **at** r for all values $u_1, \ldots, u_n$.

Region resetting

We write $H \setminus r$ for the heap resulting after *resetting* region r in H, that is

$$(H \setminus r)(a) \overset{\text{def}}{=} \begin{cases} H(a) & \text{if } a = (r', o) \wedge r' \neq r \\ \text{undefined} & \text{otherwise} \end{cases}$$

Thus, $\text{dom}(H \setminus r) = \text{dom}(H) \setminus \{(r, o) : o \in \textbf{Offset}\}$. Region resetting will be used only in the box scheduler semantics of Section 6.3.

6.2.2 Preliminary definitions

Stack

A *stack value* is either an unboxed value or a *continuation*, i.e. the runtime representation of the function call context. The *stack* is a sequence of stack values. The empty stack is [] and $s : S'$ is a stack with s on top of S'.

$$\begin{array}{rcl} s & \in & \textbf{Stackval} \\ s & ::= & u \mid \kappa \\ S & \in & \textbf{Stack} \overset{\text{def}}{=} \textbf{Stackval}^* \\ S & ::= & [] \mid s : S \end{array}$$

Environment

The *environment* is a (finite) map from identifiers to unboxed values:

$$E \in \textbf{Env} \overset{\text{def}}{=} \textbf{Var} \rightarrow_{\text{fin}} \textbf{Unboxed}$$

The environment is used to associate values to free identifiers. In a compiled code implementation the environment would exist only at compile-time and references to identifiers would be translated into references to stack offsets.

Control directives

The *control* component is a sequence of directives specifying the pending evaluation.

$$\begin{array}{rcll}
C & \in & \mathbf{Control} \overset{\text{def}}{=} \mathbf{Dir}^* & \text{sequence of directives} \\
C & ::= & [] \quad | \quad d : C & \\
 & & & \\
d & \in & \mathbf{Dir} & \\
d & ::= & \mathsf{eval}(e) & \text{evaluate expression} \\
 & | & \mathsf{bind}(x_1, \ldots, x_n) & \text{extend environment } (n \geq 0) \\
 & | & \mathsf{fbind}(x_1, \ldots, x_n) & \text{reset environment } (n \geq 0) \\
 & | & \mathsf{ret}(n) & \text{dynamic return } (n \geq 0) \\
 & | & \mathsf{sret}(n, E) & \text{static return } (n \geq 0) \\
 & | & \mathsf{select}(\mathit{alts}) & \text{pattern selection} \\
 & | & \mathsf{mkcons}(c, n) & \text{make constructor } (n \geq 0) \\
 & | & \mathsf{succ} & \text{primitive operation}
\end{array}$$

The initial control for evaluating a closed expression e is $\mathsf{eval}(e) : []$; the rules of Table 6.1 perform single-step transitions replacing the evaluation of e it by directives for evaluating sub-expressions; the evaluation terminates when the control is $[]$ and the result will be left on top of the stack.

Continuations

A *continuation* is the representation of the dynamic context for function invocation. For our abstract machine, a continuation is a pair of the current control and environment:

$$\kappa \in \mathbf{Control} \times \mathbf{Env}$$

In a compiled code implementation, the continuation would be represented in the function's activation record (e.g. a return address and frame pointer).

Note that the environment can contain heap addresses; to restore the environment in a different heap, we need to ensure a safety condition: no address accessible in the environment was deallocated from the heap. In fact, our semantics will satisfy a much stronger condition: heap values are not deallocated during expression reduction (Lemma 6.1).

States

A machine state σ is a 4-tuple $\langle C, E, S, H \rangle$. The set of all states is Σ.

$$\sigma \in \Sigma \overset{\text{def}}{=} \mathbf{Control} \times \mathbf{Env} \times \mathbf{Stack} \times \mathbf{Heap}$$

We use two projection functions $\mathsf{stack} : \Sigma \to \textbf{Stack}$ and $\mathsf{heap} : \Sigma \to \textbf{Heap}$ to refer to the stack and heap components of a state.

Heap and stack metrics

We take the storage required for an unboxed value as our unit of cost. In a real implementation, this typically corresponds to one machine word. The storage required for a boxed value with n arguments $\langle c, u_1 \ldots, u_n\rangle$ should then be proportional to n. Assuming a contiguous memory layout using one extra word for the tag c and the number of fields n, we define $|\langle c, u_1 \ldots, u_n\rangle| = 1 + n$. The total storage required for a heap H is then $|H| = \sum_{a \in \mathrm{dom}(H)} |H(a)|$, again assuming contiguous allocation.

The storage size for a stack S is simply the length of S, i.e. $|[]| = 0$ and $|s : S| = 1 + |S|$. This means that we are (arbitrarily) considering that each continuation requires the same storage as an unboxed value.

It would be straightforward to assume different storage overheads for either heap or stack values and modify the development accordingly. Alternatively, we could derive a parametric analysis where costs are expressed as multiples of some basic constants. We refrain from doing so to avoid burdening the presentation. Furthermore, we will develop an analysis that is (to some extent) independent of the specific cost model by means of cost annotations (see Section 6.4.4).

6.2.3 Transition rules

The transition rules for the Core Hume machine are presented in Table 6.1 in the style of structural operational semantics (Plotkin 1981). Each transition has the form $\sigma \overset{r}{\hookrightarrow} \sigma'$, meaning that the machine proceeds in a single-step from state σ to state σ' performing allocations in region r. Transitions are specified with respect to the program P, i.e. a set of global function definitions. Some rules have preconditions and utilise Haskell-style pattern matching. We describe each of the rules informally:

- Rules (6.1) and (6.2) specify evaluation of integers and identifiers; in both cases the corresponding unboxed value is pushed onto the stack.

- Rule (6.3) specifies evaluation of function application: 1) the current context is captured in a continuation and pushed onto the stack; 2) the arguments are evaluated in reverse order; 3) the formal parameters are bound in an empty environment; 4) the function body is evaluated; finally, 5) the return directive restores the evaluation context.

- Rule (6.4) specifies let-evaluation as an inlined one-argument function. Unlike

$$\langle \mathsf{eval}(n) : C,\ E,\ S,\ H\rangle \overset{r}{\hookrightarrow} \langle C,\ E,\ n : S,\ H\rangle \tag{6.1}$$

$$\langle \mathsf{eval}(x) : C,\ E,\ S,\ H\rangle \overset{r}{\hookrightarrow} \langle C,\ E,\ E(x) : S,\ H\rangle \tag{6.2}$$

$$\frac{(f\ (x_1,\ldots,x_n) = e') \in P}{\begin{array}{r}\langle \mathsf{eval}(f\ (e_1,\ldots,e_n) : C),\ E,\ S,\ H\rangle \overset{r}{\hookrightarrow} \langle \mathsf{eval}(e_n) : \ldots : \mathsf{eval}(e_1) : \qquad\qquad \\ \mathsf{fbind}(x_1,\ \ldots,\ x_n) : \mathsf{eval}(e') : \mathsf{ret}(n) : [],\ E,\ \langle C,\ E\rangle : S,\ H\rangle\end{array}} \tag{6.3}$$

$$\begin{array}{l}\langle \mathsf{eval}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) : C,\ E,\ S,\ H\rangle \overset{r}{\hookrightarrow} \\ \qquad\qquad \langle \mathsf{eval}(e_1) : \mathsf{bind}(x) : \mathsf{eval}(e_2) : \mathsf{sret}(1, E) : C,\ E,\ S,\ H\rangle\end{array} \tag{6.4}$$

$$\langle \mathsf{eval}(\mathsf{case}\ e\ \mathsf{of}\ \mathit{alts}) : C,\ E,\ S,\ H\rangle \overset{r}{\hookrightarrow} \langle \mathsf{eval}(e) : \mathsf{select}(\mathit{alts}) : C,\ E,\ S,\ H\rangle \tag{6.5}$$

$$\begin{array}{l}\langle \mathsf{eval}(c\ (e_1,\ldots,e_n)) : C,\ E,\ S,\ H\rangle \overset{r}{\hookrightarrow} \\ \qquad\qquad \langle \mathsf{eval}(e_n) : \ldots : \mathsf{eval}(e_1) : \mathsf{mkcons}(c, n) : C,\ E,\ S,\ H\rangle\end{array} \tag{6.6}$$

$$\langle \mathsf{eval}(\mathsf{succ}\ e) : C,\ E,\ S,\ H\rangle \overset{r}{\hookrightarrow} \langle \mathsf{eval}(e) : \mathsf{succ} : C,\ E,\ S,\ H\rangle \tag{6.7}$$

$$\begin{array}{l}\langle \mathsf{bind}(x_1,\ \ldots,\ x_n) : C,\ E,\ u_1 : \ldots : u_n : S,\ H\rangle \overset{r}{\hookrightarrow} \\ \qquad\qquad \langle C,\ E[x_1 \mapsto u_1,\ldots,x_n \mapsto u_n],\ u_1 : \ldots : u_n : S,\ H\rangle\end{array} \tag{6.8}$$

$$\begin{array}{l}\langle \mathsf{fbind}(x_1,\ \ldots,\ x_n) : C,\ E,\ u_1 : \ldots : u_n : S,\ H\rangle \overset{r}{\hookrightarrow} \\ \qquad\qquad \langle C,\ [x_1 \mapsto u_1,\ldots,x_n \mapsto u_n],\ u_1 : \ldots : u_n : S,\ H\rangle\end{array} \tag{6.9}$$

$$\frac{H(a) = \langle c, u_1,\ldots,u_n\rangle \qquad (c\ (x_1,\ldots,x_n) \to e) \in \mathit{alts}}{\begin{array}{r}\langle \mathsf{select}(\mathit{alts}) : C,\ E,\ a : S,\ H\rangle \overset{r}{\hookrightarrow} \langle \mathsf{bind}(x_1,\ \ldots,\ x_n) : \mathsf{eval}(e) : \qquad \\ \mathsf{sret}(n, E) : C,\ E,\ u_1 : \ldots : u_n : S,\ H\rangle\end{array}} \tag{6.10}$$

$$\begin{array}{l}\langle \mathsf{ret}(n) : C,\ E,\ u_0 : u_1 : \ldots : u_n : \langle C',\ E'\rangle : S,\ H\rangle \overset{r}{\hookrightarrow} \\ \qquad\qquad\qquad\qquad\qquad\qquad \langle C',\ E',\ u_0 : S,\ H\rangle\end{array} \tag{6.11}$$

$$\langle \mathsf{sret}(n, E') : C,\ E,\ u_0 : u_1 : \ldots : u_n : S,\ H\rangle \overset{r}{\hookrightarrow} \langle C,\ E',\ u_0 : S,\ H\rangle \tag{6.12}$$

$$\frac{S = u_1 : \ldots : u_n : S' \qquad a \notin \mathrm{dom}(H) \qquad a\ \mathbf{at}\ r \qquad \forall i.\ u_i\ \mathbf{at}\ r}{\langle \mathsf{mkcons}(c, n) : C,\ E,\ S,\ H\rangle \overset{r}{\hookrightarrow} \langle C,\ E,\ a : S',\ H[a \mapsto \langle c, u_1,\ldots,u_n\rangle]\rangle} \tag{6.13}$$

$$\langle \mathsf{succ} : C,\ E,\ n : S,\ H\rangle \overset{r}{\hookrightarrow} \langle C,\ E,\ (1 + n) : S,\ H\rangle \tag{6.14}$$

Table 6.1: Small-step operational semantics for Core Hume expressions.

rule (6.3), the control flow is statically known and therefore no continuation is pushed onto the stack; the environment is simply extended for evaluation of the body and restored by a static return.

- Rule (6.5) specifies that the evaluation of a case expression requires evaluating the discriminant and selecting one alternative. Rule (6.10) for the select directive unpacks a boxed value onto the stack, binds pattern variables and evaluates the match expressions. As in the evaluation of a let-expression, no continuation is pushed.

- Rules (6.6) and (6.13) for constructors evaluate the arguments, build a result in the heap and push its address on the stack. The preconditions 'a **at** r' and 'u_i **at** r' require that both the allocated value and arguments reside in the same region. We remark that (6.13) is the only rule that modifies the heap.

- Rules (6.7) and (6.14) evaluate a primitive operation, namely, the integer successor function. The result is an unboxed integer and therefore stored in the stack. It would be straightforward to extend the rules with other primitive operations, and we omit them from this presentation for brevity.

- Rules (6.8) and (6.9) modify the environment by binding identifiers to the values on top of the stack. Directive bind extends the existing environment while funbind creates a new environment; the former is used in let and case expressions while the latter is used in function calls. Note that the bound values are *not* removed from the stack: binding is simply the association of identifiers to stack values. Stack space is reclaimed only by a static or dynamic return when exiting a scope.

- Rule (6.11) implements a *dynamic return* at the end of a function call. The result value is left on the top of the stack and the continuation is n elements below. Therefore, the stack shrinks by n elements and the context is restored from the continuation.

- Rule (6.12) implements a *static return* when exiting the scope of a let binding or case alternative. The result value is on the top of the stack, the stack shrinks by n elements, the environment is restored and the control continues with the next directive.

We write $\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}$ for the reflexive and transitive closure of $\overset{r}{\hookrightarrow}$, i.e. $\sigma_0 \overset{r}{\hookrightarrow\!\!\!\!\rightarrow} \sigma_n$ if and only if $\exists \sigma_1 \ldots \sigma_{n-1} : \sigma_0 \overset{r}{\hookrightarrow} \sigma_1 \overset{r}{\hookrightarrow} \cdots \overset{r}{\hookrightarrow} \sigma_{n-1} \overset{r}{\hookrightarrow} \sigma_n$. When it is clear from the context, we will sometimes omit the region parameter from reduction relations, writing $\hookrightarrow$ and $\hookrightarrow\!\!\!\!\rightarrow$ instead of $\overset{r}{\hookrightarrow}$ and $\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}$.

6.2.4 Auxiliary results

Lemma 6.1 (Heap monotonicity) *If* $\sigma \overset{r}{\hookrightarrow} \sigma'$ *then* $\mathsf{heap}(\sigma) \subseteq \mathsf{heap}(\sigma')$*.*

Proof: Only rule (6.13) modifies the heap with $H' = H[a \mapsto \cdots]$; the result follows immediately from the assumption $a \notin \mathrm{dom}(H)$. □

The next lemma states a standard invariant for a stack machine: successful evaluation of an expression extends the stack by one value (the result of evaluation) and leaves the environment unchanged.

Lemma 6.2 (Evaluation) *If* $\langle \mathsf{eval}(e) : C,\, E,\, S,\, H\rangle \overset{r}{\hookrightarrow\!\!\!\!\rightarrow} \langle C,\, E',\, S',\, H'\rangle$ *then* $E' = E$ *and there exists* u *such that* $S' = u : S$*.*

Proof: By induction on the number n of reduction steps. For $n = 1$, the only applicable rules are (6.1) and (6.2) and it is immediate that the result holds. For $n > 1$, we proceed by analysis on the structure of the expression e.

If $e \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$; the transition sequence must be:

$$\langle \mathsf{eval}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) : C,\, E,\, S,\, H\rangle$$

$$\hookrightarrow \langle \mathsf{eval}(e_1) : \mathsf{bind}(x) : \mathsf{eval}(e_2) : \mathsf{sret}(1, E) : C,\, E,\, S,\, H\rangle \qquad \text{by rule (6.4)}$$

$$\hookrightarrow\!\!\!\!\rightarrow \langle \mathsf{bind}(x) : \mathsf{eval}(e_2) : \mathsf{sret}(1, E) : C,\, E,\, u_1 : S,\, H_1\rangle$$

by the induction hypothesis on e_1

$$\hookrightarrow \langle \mathsf{eval}(e_2) : \mathsf{sret}(1, E) : C,\, E',\, u_1 : S,\, H_1\rangle \qquad \text{by rule (6.8)}$$

$$\hookrightarrow\!\!\!\!\rightarrow \langle \mathsf{sret}(1, E) : C,\, E',\, u_2 : u_1 : S,\, H_2\rangle$$

by the induction hypothesis on e_2

$$\hookrightarrow \langle C,\, E,\, u_2 : S,\, H_2\rangle \qquad \text{by rule (6.12)}$$

If $e \equiv f\ (e_1, \ldots, e_m)$; the transition sequence must be:

$$\langle \mathsf{eval}(f\ (e_1, \ldots, e_m)) : C,\, E,\, S,\, H\rangle$$

$$\hookrightarrow \langle \mathsf{eval}(e_m) : \ldots : \mathsf{eval}(e_1) : \mathsf{fbind}(\vec{x}) : \mathsf{eval}(e') : \mathsf{ret}(m) : [],\, E,\, \langle C,\, E\rangle : S,\, H\rangle$$

by rule (6.3)

$$\hookrightarrow\!\!\!\!\rightarrow \langle \mathsf{fbind}(\vec{x}) : \mathsf{eval}(e') : \mathsf{ret}(m) : [],\, E,\, u_1 : \ldots : u_m : \langle C,\, E\rangle : S,\, H'\rangle$$

using the induction hypothesis m times on $e_m,\, \ldots,\, e_1$

$$\hookrightarrow \langle \mathsf{eval}(e') : \mathsf{ret}(m) : [],\, [\vec{x} \mapsto \vec{u}],\, u_1 : \ldots : u_m : \langle C,\, E\rangle : S,\, H'\rangle$$

by rule (6.9)

$$\hookrightarrow\!\!\!\!\rightarrow \langle \mathsf{ret}(m) : [],\, [\vec{x} \mapsto \vec{u}],\, u_0 : u_1 : \ldots : u_m : \langle C,\, E\rangle : S,\, H''\rangle$$

by the induction hypothesis on e'

$\hookrightarrow \langle C, E, u_0 : S, H'' \rangle$

by rule (6.11)

The proof for the case statement is similar to that for the let; the proofs for constructors and primitives follow directly from rules (6.6), (6.13), (6.7) and (6.14). This concludes the proof of Lemma 6.2. □

6.2.5 Big-step evaluation semantics

We will now define an *instrumented big-step evaluation relation* that abstracts individual reduction steps but records stack and heap usage.[1] The objective is to be able to reason about the evaluation costs compositionally, i.e. define the stack and heap costs of an expression in terms of the costs of sub-expressions. This will pave the way for the source-level analysis for stack and heap costs in Section 6.4.

The big-step judgements have the form

$$H, E \vdash e \Downarrow_r u, H', \delta, \gamma$$

meaning that e reduces to u under environment E, heap H and region r; the final heap is H' and δ and γ are, respectively, the relative stack and heap metrics used in the evaluation of e.

First, we remark that the relative heap for a reduction sequence $\sigma \hookrightarrow\!\!\!\!\rightarrow \sigma'$ is simply the difference $|\mathsf{heap}(\sigma')| - |\mathsf{heap}(\sigma)|$ because no deallocation occurs during expression reduction.

For the stack metric, however, we need to keep track of the maximum depth in all intermediate states. We do so using an instrumented version of the transitive reduction relation: $\sigma \overset{r}{\underset{M}{\hookrightarrow\!\!\!\!\rightarrow}} \sigma'$ means that M is the maximum stack depth in the reduction sequence from σ to σ'. This relation is formally defined by two inductive rules:

$$\frac{M = |\mathsf{stack}(\sigma)|}{\sigma \overset{r}{\underset{M}{\hookrightarrow\!\!\!\!\rightarrow}} \sigma} \qquad \frac{\sigma \overset{r}{\hookrightarrow} \sigma' \quad \sigma' \overset{r}{\underset{M'}{\hookrightarrow\!\!\!\!\rightarrow}} \sigma'' \quad M = \max(|\mathsf{stack}(\sigma)|, M')}{\sigma \overset{r}{\underset{M}{\hookrightarrow\!\!\!\!\rightarrow}} \sigma''}$$

Using $\overset{r}{\underset{M}{\hookrightarrow\!\!\!\!\rightarrow}}$ we now define the big-step evaluation relation.

[1] This can be seen as the formalisation of a simple profiler that records maximum usage for a whole expression.

Definition 6.3 (Big-step evaluation semantics) *The big-step evaluation relation is defined by*

$$\begin{aligned}
&H,\, E \vdash e \Downarrow_r u,\, H',\, \delta,\, \gamma \overset{def}{\iff} \\
&\qquad \forall C\ \forall S\ \langle \mathsf{eval}(e) : C,\, E,\, S,\, H\rangle \underset{M}{\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}} \langle C,\, E,\, u : S,\, H'\rangle \\
&\qquad\qquad \wedge\, \delta = M - |S| \wedge \gamma = |H'| - |H|\ .
\end{aligned}$$

Two remarks should be made regarding this definition. First, note that the stack size of the final state is $1 + |S|$; therefore $M \geq 1 + |S|$ which implies that $\delta \geq 1$.

Second, the universal quantification in the code C and stack S makes big-step evaluation independent of context. This allows us to derive structural rules for big step evaluation with stack and heap costs (Table 6.2).

Lemma 6.4 *The rules of Table 6.2 are admissible.*

Proof: The proofs are straightforward but tedious. We prove the rules for the let expression, function application and case expression in detail.

Let-expression: the hypotheses of rule (6.17) are

$$H_0,\, E \vdash e_1 \Downarrow_r u_1,\, H_1,\, \delta_1,\, \gamma_1 \tag{6.22}$$

$$H_1,\, E[x \mapsto u_1] \vdash e_2 \Downarrow_r u_2,\, H_2,\, \delta_2,\, \gamma_2 \tag{6.23}$$

and we want to conclude

$$H_0,\, E \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow_r u_2,\, H_2,\, \max(\delta_1,\, 1 + \delta_2),\, \gamma_1 + \gamma_2 \tag{6.24}$$

Let C and S be arbitrary; the reduction sequence is as follow:

$$\begin{aligned}
&\langle \mathsf{eval}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) : C,\, E,\, S,\, H_0\rangle \\
\overset{r}{\hookrightarrow}\ &\langle \mathsf{eval}(e_1) : \mathsf{bind}(x) : \mathsf{eval}(e_2) : \mathsf{sret}(1, E) : C,\, E,\, S,\, H_0\rangle \\
\underset{M_1}{\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}}\ &\langle \mathsf{bind}(x) : \mathsf{eval}(e_2) : \mathsf{sret}(1, E) : C,\, E,\, u_1 : S,\, H_1\rangle \\
\overset{r}{\hookrightarrow}\ &\langle \mathsf{eval}(e_2) : \mathsf{sret}(1, E) : C,\, E[x \mapsto u_1],\, u_1 : S,\, H_1\rangle \\
\underset{M_2}{\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}}\ &\langle \mathsf{sret}(1, E) : C,\, E[x \mapsto u_1],\, u_2 : u_1 : S,\, H_2\rangle \\
\overset{r}{\hookrightarrow}\ &\langle C,\, E,\, u_2 : S,\, H_2\rangle
\end{aligned}$$

Designate by δ the relative stack metric for the above evaluation. By definition of $\underset{M}{\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}}$, we know $M_1 \geq |u_1 : S|$ and $M_2 \geq |u_2 : u_1 : S|$, and therefore

$$\begin{aligned}
\delta &= \max\{|S|,\, M_1,\, |u_1 : S|,\, M_2,\, |u_2 : u_1 : S|,\, |u_2 : S|\} - |S| \\
&= \max(M_1,\, M_2) - |S| = \max(M_1 - |S|,\, M_2 - |S|)
\end{aligned} \tag{6.25}$$

$$H,\, E \vdash n \Downarrow_r n,\, H,\, 1,\, 0 \tag{6.15}$$

$$H,\, E \vdash x \Downarrow_r E(x),\, H,\, 1,\, 0 \tag{6.16}$$

$$\frac{H_0,\, E \vdash e_1 \Downarrow_r u_1,\, H_1,\, \delta_1,\, \gamma_1 \qquad H_1,\, E[x \mapsto u_1] \vdash e_2 \Downarrow_r u_2,\, H_2,\, \delta_2,\, \gamma_2}{H_0,\, E \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow_r u_2,\, H_2,\, \max(\delta_1,\, 1 + \delta_2),\, \gamma_1 + \gamma_2} \tag{6.17}$$

$$\frac{\begin{array}{c} H_{n-i},\, E \vdash e_i \Downarrow_r u_i,\, H_{n+1-i},\, \delta_i,\, \gamma_i \qquad\qquad (1 \le i \le n) \\ H_n,\, [x_1 \mapsto u_1, \ldots, x_n \mapsto u_n] \vdash e' \Downarrow_r u',\, H_{n+1},\, \delta',\, \gamma' \\ \delta = \max\{n + 1 - i + \delta_i\}_{i=1}^{n} \qquad \gamma = \sum_{i=1}^{n} \gamma_i \end{array}}{H_0,\, E \vdash f\ (e_1, \ldots, e_n) \Downarrow_r u',\, H_{n+1},\, \max(\delta,\, n + 1 + \delta'),\, \gamma + \gamma'} \tag{6.18}$$
$$\text{if } (f\ (x_1, \ldots, x_n) = e') \in P$$

$$\frac{\begin{array}{c} (\forall i) \qquad H_{n-i},\, E \vdash e_i \Downarrow_r u_i,\, H_{n+1-i},\, \delta_i,\, \gamma_i \qquad u_i\ \mathbf{at}\ r \\ a \notin \mathrm{dom}(H_n) \qquad a\ \mathbf{at}\ r \qquad \delta = \max\{n - i + \delta_i\}_{i=1}^{n} \qquad \gamma = \sum_{i=1}^{n} \gamma_i \end{array}}{H_0,\, E \vdash c\ (e_1, \ldots, e_n) \Downarrow_r a,\, H_n[a \mapsto \langle c, u_1, \ldots, u_n \rangle],\, \delta,\, \gamma + 1 + n} \tag{6.19}$$

$$\frac{\begin{array}{c} H_0,\, E \vdash e \Downarrow_r a,\, H_1,\, \delta_1,\, \gamma_1 \\ H_1(a) = \langle c, u_1, \ldots, u_n \rangle \qquad (c\ (x_1, \ldots, x_n) \to e') \in \mathit{alts} \\ H_1,\, E[x_1 \mapsto u_1, \ldots, x_n \mapsto u_n] \vdash e' \Downarrow_r u',\, H_2,\, \delta_2,\, \gamma_2 \end{array}}{H_0,\, E \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathit{alts} \Downarrow_r u',\, H_2,\, \max(\delta_1, n + \delta_2),\, \gamma_1 + \gamma_2} \tag{6.20}$$

$$\frac{H,\, E \vdash e \Downarrow_r n,\, H',\, \delta,\, \gamma}{H,\, E \vdash \mathsf{succ}\ e \Downarrow_r n + 1,\, H',\, \delta,\, \gamma} \tag{6.21}$$

Table 6.2: Rules for big-step evaluation.

By hypotheses (6.22) and (6.23), we obtain relations between the maximum and relative stack depths for subexpressions e_1 and e_2:

$$\begin{aligned}
\delta_1 &= M_1 - |S| \\
\delta_2 &= M_2 - |u_1 : S| = M_2 - (1 + |S|) \\
\iff 1 + \delta_2 &= M_2 - |S|
\end{aligned}$$

Replacing the above equalities in (6.25) we get $\delta = \max(\delta_1,\, 1+\delta_2)$, thus establishing the relative stack result. For the relative heap result, we remark that $\gamma_1 = |H_1| - |H_0|$ and $\gamma_2 = |H_2| - |H_1|$ and therefore

$$\gamma = |H_2| - |H_0| = \underbrace{|H_1| - |H_0|}_{\gamma_1} + \underbrace{|H_2| - |H_1|}_{\gamma_2} = \gamma_1 + \gamma_2 \ .$$

Function application: the hypotheses of rule (6.18) are

$$H_{n-i},\, E \vdash e_i \Downarrow_r u_i,\, H_{n+1-i},\, \delta_i,\, \gamma_i \qquad (1 \le i \le n) \tag{6.26}$$

$$H_n,\, [x_1 \mapsto u_1, \ldots, x_n \mapsto u_n] \vdash e' \Downarrow_r u',\, H_{n+1},\, \delta',\, \gamma' \tag{6.27}$$

and we want to conclude

$$H_0,\, E \vdash f\ (e_1, \ldots, e_n) \Downarrow_r u',\, H_{n+1},\, \max(\delta,\, n+1+\delta'),\, \gamma + \gamma' \tag{6.28}$$

where $\delta = \max\{n+1-i+\delta_i\}_{i=1}^{n}$ and $\gamma = \sum_{i=1}^{n} \gamma_i$. Let C and S be arbitrary; the reduction sequence for the application is as follows:

$$\begin{aligned}
&\langle \mathsf{eval}(f\ (e_1, \ldots, e_n) : C),\, E,\, S,\, H_0 \rangle \\
\overset{r}{\hookrightarrow}\ &\langle \mathsf{eval}(e_n) : \ldots : \mathsf{eval}(e_1) : \mathsf{fbind}(\vec{x}) : \mathsf{eval}(e') : \mathsf{ret}(n) : [],\, E,\, \langle C,\, E \rangle : S,\, H_0 \rangle \\
\underset{M_n}{\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}}\ &\langle \mathsf{eval}(e_{n-1}) : \ldots : \mathsf{eval}(e_1) : \mathsf{fbind}(\vec{x}) : \mathsf{eval}(e') : \mathsf{ret}(n) : [],\, E, \\
&\quad u_n : \langle C,\, E \rangle : S,\, H_1 \rangle \\
&\vdots \\
\underset{M_2}{\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}}\ &\langle \mathsf{eval}(e_1) : \mathsf{fbind}(\vec{x}) : \mathsf{eval}(e') : \mathsf{ret}(n) : [],\, E,\, u_2 : \ldots : u_n : \langle C,\, E \rangle : S,\, H_{n-1} \rangle \\
\underset{M_1}{\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}}\ &\langle \mathsf{fbind}(\vec{x}) : \mathsf{eval}(e') : \mathsf{ret}(n) : [],\, E,\, u_1 : \ldots : u_n : \langle C,\, E \rangle : S,\, H_n \rangle \\
\overset{r}{\hookrightarrow}\ &\langle \mathsf{eval}(e') : \mathsf{ret}(n) : [],\, [\vec{x} \mapsto \vec{u}],\, u_1 : \ldots : u_n : \langle C,\, E \rangle : S,\, H_n \rangle \\
\underset{M'}{\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}}\ &\langle \mathsf{ret}(n) : [],\, [\vec{x} \mapsto \vec{u}],\, u' : u_1 : \ldots : u_n : \langle C,\, E \rangle : S,\, H_{n+1} \rangle \\
\overset{r}{\hookrightarrow}\ &\langle C,\, E,\, u' : S,\, H_{n+1} \rangle
\end{aligned}$$

The relative stack metric for the application is

$$\begin{aligned}
\delta &= \max\{M_1,\, M_2,\, \ldots,\, M_n,\, M'\} - |S| \\
&= \max(\max\{M_i - |S|\}_{i=1}^{n},\, M' - |S|)
\end{aligned} \tag{6.29}$$

Using hypotheses (6.26) and (6.27), we obtain:

$$\delta_i = M_i - |u_{i+1} : \ldots : u_n : \langle C,\, E\rangle : S| = M_i - (n + 1 - i + |S|) \tag{6.30}$$

$$\delta' = M' - |u_1 : \ldots : u_n : \langle C,\, E\rangle : S| = M' - (n + 1 + |S|) \tag{6.31}$$

Isolating δ_i and δ' in the above equations yields

$$M_i - |S| = \delta_i + n + 1 - i \tag{6.32}$$

$$M' - |S| = \delta' + n + 1 \tag{6.33}$$

Replacing the above equations in (6.29) we get the stack result:

$$\delta = \max(\max\{\delta_i + n + 1 - i\}_{i=1}^{n},\, \delta' + n + 1) \tag{6.34}$$

The heap result is immediate because

$$\begin{aligned} |H_{n+1}| - |H_0| &= \underbrace{|H_{n+1}| - |H_n|}_{\gamma'} + \underbrace{|H_n| - |H_{n-1}|}_{\gamma_n} + \cdots + \underbrace{|H_1| - |H_0|}_{\gamma_1} \\ &= \gamma' + \sum_{i=1}^{n} \gamma_i \end{aligned}$$

Case expression: the hypotheses of rule (6.20) are

$$H_0,\, E \vdash e \Downarrow_r a,\, H_1,\, \delta_1,\, \gamma_1 \tag{6.35}$$

$$H_1(a) = \langle c, u_1, \ldots, u_n\rangle \tag{6.36}$$

$$(c\ (x_1, \ldots, x_n) \to e') \in \mathit{alts} \tag{6.37}$$

$$H_1,\, E[x_1 \mapsto u_1, \ldots, x_n \mapsto u_n] \vdash e' \Downarrow_r u',\, H_2,\, \delta_2,\, \gamma_2 \tag{6.38}$$

and the conclusion is

$$H_0,\, E \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathit{alts} \Downarrow_r u',\, H_2,\, \max(\delta_1, n + \delta_2),\, \gamma_1 + \gamma_2 \tag{6.39}$$

Starting from an arbitrary code C and stack S, the reduction of the case expression is:

$$\begin{aligned} & \langle \mathsf{eval}(\mathsf{case}\ e\ \mathsf{of}\ \mathit{alts}) : C,\, E,\, S,\, H_0\rangle \\ \overset{r}{\hookrightarrow}\ & \langle \mathsf{eval}(e) : \mathsf{select}(\mathit{alts}) : C,\, E,\, S,\, H_0\rangle \\ \underset{M_1}{\overset{r}{\hookrightarrow\!\!\!\rightarrow}}\ & \langle \mathsf{select}(\mathit{alts}) : C,\, E,\, a : S,\, H_1\rangle \\ \overset{r}{\hookrightarrow}\ & \langle \mathsf{bind}(x_1,\, \ldots,\, x_n) : \mathsf{eval}(e') : \mathsf{sret}(n, E) : C,\, E,\, u_1 : \ldots : u_n : S,\, H_1\rangle \\ \overset{r}{\hookrightarrow}\ & \langle \mathsf{eval}(e') : \mathsf{sret}(n, E) : C,\, E[x_1 \mapsto u_1,\, \ldots,\, x_n \mapsto u_n],\, u_1 : \ldots : u_n : S,\, H_1\rangle \\ \underset{M_2}{\overset{r}{\hookrightarrow\!\!\!\rightarrow}}\ & \langle \mathsf{sret}(n, E) : C,\, E[x_1 \mapsto u_1,\, \ldots,\, x_n \mapsto u_n],\, u' : u_1 : \ldots : u_n : S,\, H_2\rangle \\ \overset{r}{\hookrightarrow}\ & \langle C,\, E,\, u' : S,\, H_2\rangle \end{aligned}$$

Using hypothesis (6.35) and (6.38) we obtain:

$$\begin{aligned} & \delta_1 = M_1 - |S| \\ & \delta_2 = M_2 - |u_1 : \ldots : u_n : S| = M_2 - (|S| + n) \\ \iff \; & n + \delta_2 = M_2 - |S| \end{aligned}$$

The relative stack metric δ for the case expression is then

$$\begin{aligned} \delta &= \max\{|S|,\, |S|+1,\, |S|+n,\, M_1,\, M_2\} - |S| \\ &= \max\{1,\, n,\, M_1 - |S|,\, M_2 - |S|\} \\ &= \max\{1,\, n,\, \delta_1,\, n + \delta_2\} \\ &= \max(\delta_1,\, n + \delta_2) \end{aligned}$$

since $\delta_1 \geq 1$ and $\delta_2 \geq 1$ (see the remark following Definition 6.3). The relative heap result is again immediate because

$$\gamma = |H_2| - |H_0| = (\underbrace{|H_2| - |H_1|}_{\gamma_2}) + (\underbrace{|H_1| - |H_0|}_{\gamma_1}) = \gamma_1 + \gamma_2$$

The proofs for the rules (6.15) (6.16) (6.21) for constants, variables and the primitive successor follow directly from the single-step reduction; we therefore omit them. The proof for the constructor application rule (6.19) is similar to the proof of rule (6.18) for function application. This concludes the proof of Lemma 6.4. □

Whenever we do not care about the stack and heap metrics we will omit them from the reduction relation, writing $H,\, E \vdash e \Downarrow_r u,\, H'$ instead of $\exists\delta\ \exists\gamma\ H,\, E \vdash e \Downarrow_r u,\, H',\, \delta,\, \gamma$. We will also introduce a notation for a sequence of big-step evaluations, each producing a result in a distinct region. This notation will be used for the evaluation of the right-hand of a box rule:

$$\begin{aligned} & H_1,\, E \vdash (e_1, \ldots, e_n) \Downarrow_{(r_1,\ldots,r_n)} (u_1, \ldots, u_n),\ H_{n+1} \overset{\text{def}}{\iff} \\ & \qquad\qquad H_i,\, E \vdash e_i \Downarrow_{r_i} u_i,\, H_{i+1} \wedge u_i \ \textbf{at}\ r_i \quad (1 \leq i \leq n) \qquad (6.40) \end{aligned}$$

Note that the threading of the heap from H_1 to H_{n+1} in the above definition fixes the order of evaluation from e_1 to e_n. Since the output regions $r_1, \ldots, r_n$ are distinct from each other and from the input wire regions, no real data dependency exists between the evaluations.

6.3 Operational semantics for boxes

A Core Hume program defines a network of communicating processes as a sequence of box, wire, data type and functions declarations. In the previous section we defined an abstract machine for evaluation of expressions and functions.

In this section we will define the operational semantics for the coordination layer of a Core Hume program in the form of a single-step transition relation on network states. This semantics implements the following actions:

- scheduling the next box with sufficient available inputs;
- synchronisation of accesses to wires, i.e. keep record of which wires are available and ensure boxes only read/write to available wires;
- memory management for the communication wires, i.e. reclaim heap space for wires after inputs are consumed.

We will use the big-step evaluation relation defined in Section 6.2.5 to evaluate expressions in the right-hand side of box rules. This means that expression evaluation is an atomic action that cannot be preempted by the scheduler.

6.3.1 Wire values and locations

We assume that each wire is assigned a sequential index at compile-time; the set of all wire indices is denoted by **Wire**. We then define a *wire value* map as the association of each wire to a (possibly null) value:

$$\nu \in \mathbf{Wireval} \overset{\text{def}}{=} \mathbf{Wire} \to \mathbf{Unboxed} \cup \{\mathsf{null}\}$$

A wire value is either an unboxed value (defined in the expression semantics of Section 6.2) or a special null value: if $\nu(w) = \mathsf{null}$ then wire w is not available for reading (and, conversely, is available for writing).

Values in wires result from evaluating arbitrary Hume expressions (e.g. tuples, lists, etc.) and might therefore require heap storage. Since values in wires might be written and consumed independently, we will use separate heap regions for each wire. We introduce two *wire location* maps

$$\rho,\, \tilde{\rho} \in \mathbf{Wireloc} \overset{\text{def}}{=} \mathbf{Wire} \to \mathbf{Region}$$

to associate each wire w with two distinct heap regions: $\rho(w)$ is associated with the value of wire w in the *current* scheduler iteration and $\tilde{\rho}(w)$ with the value in the *next* iteration.

The combination of wire value and region maps allows an efficient and predictable recycling of heap space:

- to consume the value on a wire w we set $\nu(w) \leftarrow \mathsf{null}$ and reset region $\rho(w)$;

- the output value in a wire w' is constructed in the region for the next iteration $\tilde{\rho}(w')$;
- when all outputs $u_1, \ldots, u_n$ to wires $w'_1, \ldots, w'_n$ are available, the values are updated in single step by setting $\nu(w'_i) \leftarrow u_i$ and exchanging $\rho(w_i)$ and $\tilde{\rho}(w_i)$.[2]

These operations can be implemented by simple array updates, thus allowing heap reuse at the end of a scheduling step with no space overhead and in predictable time.

Finally, since each wire is assigned two distinct regions, the total number of regions is twice the number of wires, and therefore known at compile-time.

6.3.2 Runtime configuration of boxes

The runtime configuration for a box is a 6-tuple of a *state*, a sequence of pending *output values*, a *matching* flag, a sequence of *box rules* and sequences of *input* and *output wires*. The state of a box is either *ready* (waiting to read inputs) or *blocked* (waiting to write outputs). The matching flag is either *fair* or *unfair*; fair matching implies re-ordering alternatives after a successful match.

$$\begin{aligned}
\mathbf{Box} &\stackrel{\text{def}}{=} \mathbf{BoxState} \times \mathbf{Unboxed}^* \times \mathbf{Match} \times \mathbf{Alts} \times \mathbf{Wire}^* \times \mathbf{Wire}^* \\
\mathbf{BoxState} &\stackrel{\text{def}}{=} \{\mathsf{ready}, \mathsf{blocked}\} \\
\mathbf{Match} &\stackrel{\text{def}}{=} \{\mathsf{fair}, \mathsf{unfair}\}
\end{aligned}$$

The state of the network of boxes is a 5-tuple consisting of a sequence of *box configurations*, a *heap* and the *wire value* and *location* maps.

$$\mathbf{Coord} \stackrel{\text{def}}{=} \mathbf{Box}^* \times \mathbf{Heap} \times \mathbf{Wireval} \times \mathbf{Wireloc} \times \mathbf{Wireloc}$$

6.3.3 Scheduler

Boxes are scheduled in a circular queue, following the order of declarations in the program. To determine whether to run a box, each of its rules is tried in order. When a match is found, the right-hand side of the matching rule is evaluated and the outputs are written to the output wires. The scheduler then moves the box to end of the queue and proceeds to the next box. Scheduling is non-preemptive: when a rule match is found, the corresponding right-hand-side expression runs to completion. This means that expression evaluation is an atomic action from the point-of-view of the scheduler.

[2] This exchange operation is equivalent to two simultaneous *region-renaming* operations of Henglein et al. (2001).

$$\frac{\begin{array}{r}H \vdash \nu(in)\ \mathbf{match}\ \mathit{rules} \Rightarrow (\vec{ap} \to \vec{ae}),\, E \qquad H,\, E \vdash \vec{ae} \Downarrow_{\tilde{\rho}(out)} \vec{u},\, H' \\ \nu' = \nu[\{in_i \mapsto \mathsf{null} \mid ap_i \neq *\}] \qquad H'' = H' \setminus \{\rho(in_i) \mid ap_i \neq *\}\end{array}}{\begin{array}{l}\langle(\mathsf{ready}, [], \mathsf{unfair}, \mathit{rules},\, in,\, out) : \mathit{boxes},\, H,\, \nu,\, \rho,\, \tilde{\rho}\rangle \curvearrowright \\ \qquad\qquad \langle(\mathsf{blocked}, \vec{u}, \mathsf{unfair}, \mathit{rules},\, in,\, out) : \mathit{boxes},\, H'',\, \nu',\, \rho,\, \tilde{\rho}\rangle\end{array}} \tag{6.41}$$

$$\frac{\begin{array}{r}H \vdash \nu(in)\ \mathbf{match}\ \mathit{rules} \Rightarrow (\vec{ap} \to \vec{ae}),\, E \qquad H,\, E \vdash \vec{ae} \Downarrow_{\tilde{\rho}(out)} \vec{u},\, H' \\ \nu' = \nu[\{in_i \mapsto \mathsf{null} \mid ap_i \neq *\}] \qquad H'' = H' \setminus \{\rho(in_i) \mid ap_i \neq *\} \\ \mathit{rules}' = (\mathit{rules} \setminus (\vec{ap} \to \vec{ae})) +\!\!+ (\vec{ap} \to \vec{ae})\end{array}}{\begin{array}{l}\langle(\mathsf{ready}, [], \mathsf{fair}, \mathit{rules},\, in,\, out) : \mathit{boxes},\, H,\, \nu,\, \rho,\, \tilde{\rho}\rangle \curvearrowright \\ \qquad\qquad \langle(\mathsf{blocked}, \vec{u}, \mathsf{fair}, \mathit{rules}',\, in,\, out) : \mathit{boxes},\, H'',\, \nu',\, \rho,\, \tilde{\rho}\rangle\end{array}} \tag{6.42}$$

$$\frac{H \nvdash \nu(in)\ \mathbf{match}\ \mathit{rules}}{\begin{array}{l}\langle(\mathsf{ready}, [], \mathit{match}, \mathit{rules},\, in,\, out) : \mathit{boxes},\, H,\, \nu,\, \rho,\, \tilde{\rho}\rangle \curvearrowright \\ \qquad\qquad \langle \mathit{boxes} +\!\!+ (\mathsf{ready}, [], \mathit{match}, \mathit{rules},\, in,\, out),\, H,\, \nu,\, \rho,\, \tilde{\rho}\rangle\end{array}} \tag{6.43}$$

$$\frac{\begin{array}{c}(\forall i)\ u_i = \mathsf{null} \vee \nu(out_i) = \mathsf{null} \qquad \nu' = \nu[\{out_i \mapsto u_i \mid u_i \neq \mathsf{null}\}] \\ \rho' = \rho[\{out_i \mapsto \tilde{\rho}(out_i) \mid u_i \neq \mathsf{null}\}] \qquad \tilde{\rho}' = \tilde{\rho}[\{out_i \mapsto \rho(out_i) \mid u_i \neq \mathsf{null}\}]\end{array}}{\begin{array}{l}\langle(\mathsf{blocked}, \vec{u}, \mathit{match}, \mathit{rules},\, in,\, out) : \mathit{boxes},\, H,\, \nu,\, \rho,\, \tilde{\rho}\rangle \curvearrowright \\ \qquad\qquad \langle \mathit{boxes} +\!\!+ (\mathsf{ready}, [], \mathit{match}, \mathit{rules},\, in,\, out),\, H,\, \nu',\, \rho',\, \tilde{\rho}'\rangle\end{array}} \tag{6.44}$$

$$\frac{(\exists i)\ u_i \neq \mathsf{null} \wedge \nu(out_i) \neq \mathsf{null}}{\begin{array}{l}\langle(\mathsf{blocked}, \vec{u}, \mathit{match}, \mathit{rules},\, in,\, out) : \mathit{boxes},\, H,\, \nu,\, \rho,\, \tilde{\rho}\rangle \curvearrowright \\ \qquad\qquad \langle \mathit{boxes} +\!\!+ (\mathsf{blocked}, \vec{u}, \mathit{match}, \mathit{rules},\, in,\, out),\, H,\, \nu,\, \rho,\, \tilde{\rho}\rangle\end{array}} \tag{6.45}$$

Table 6.3: Box scheduling relation.

$$\boxed{H \vdash u \ \mathbf{match}\ ap \Rightarrow E}$$

$$H \vdash u \ \mathbf{match}\ \mathsf{*} \Rightarrow \emptyset \tag{6.46}$$

$$H \vdash u \ \mathbf{match}\ \_\mathsf{*} \Rightarrow \emptyset \tag{6.47}$$

$$\frac{u \neq \mathsf{null}}{H \vdash u \ \mathbf{match}\ \_ \Rightarrow \emptyset} \tag{6.48}$$

$$\frac{u \neq \mathsf{null}}{H \vdash u \ \mathbf{match}\ x \Rightarrow [x \mapsto u]} \tag{6.49}$$

$$\frac{a \in \mathbf{Addr} \qquad H(a) = \langle c, u_1, \ldots, u_n \rangle}{H \vdash a \ \mathbf{match}\ (c\, x_1 \ldots x_n) \Rightarrow [x_1 \mapsto u_1, \ldots, x_n \mapsto u_n]} \tag{6.50}$$

$$\boxed{H \vdash \vec{u} \ \mathbf{match}\ \vec{ap} \Rightarrow E}$$

$$\frac{H \vdash u_i \ \mathbf{match}\ ap_i \Rightarrow E_i \qquad (1 \leq i \leq n)}{H \vdash (u_1, \ldots, u_n) \ \mathbf{match}\ (ap_1, \ldots, ap_n) \Rightarrow \bigcup_{i=1}^{n} E_i} \tag{6.51}$$

$$\boxed{H \vdash \vec{u} \ \mathbf{match}\ \mathit{rules} \Rightarrow \mathit{rule},\, E}$$

$$\frac{H \vdash \vec{u} \ \mathbf{match}\ \vec{ap} \Rightarrow E}{H \vdash \vec{u} \ \mathbf{match}\ (\vec{ap} \rightarrow \vec{ae} \ \mid\ \mathit{rules}) \Rightarrow (\vec{ap} \rightarrow \vec{ae}\,),\, E} \tag{6.52}$$

$$\frac{H \nvdash \vec{u} \ \mathbf{match}\ \vec{ap} \qquad H \vdash \vec{u} \ \mathbf{match}\ \mathit{rules} \Rightarrow \mathit{rule},\, E}{H \vdash \vec{u} \ \mathbf{match}\ (\vec{ap} \rightarrow \vec{ae} \ \mid\ \mathit{rules}) \Rightarrow \mathit{rule},\, E} \tag{6.53}$$

Table 6.4: Box pattern and rule matching.

A box rule $(ap_1, \ldots, ap_n) \rightarrow (ae_1, \ldots, ae_m)$ is *runnable* if all the non-ignored inputs (i.e. with pattern other than '*') are available and match the corresponding patterns ap_i. The matching relation is specified in Table 6.4. The scheduler is specified in Table 6.3 as a single-step transition relation $\curvearrowright$ between network states. Each transition either *runs* or *blocks* the next box depending on availability of its inputs.

- Rule (6.41) specifies a successful *unfair* match: 1) check if the inputs values are available and bind pattern variables; 2) set the input values to null; 3) reset the input wire regions; 4) evaluate the right-hand-side expressions; 5) leave the box temporarily blocked (to attempt writing the outputs on the next scheduling step).

- Rule (6.42) specifies *fair* matching; except for a final re-ordering of the box rules it is identical to (6.41).

- Rule (6.43) deals with an unsuccessful match: the box is simply moved to the end of the scheduling queue.

- Rule (6.44) *unblocks* a box: 1) check if the outputs wires are available: 2) write the output values to the output wires; 3) exchange the current and next generation regions; 4) move the box to the end of the scheduling queue.

- Rule (6.45) applies when a box must remain blocked because the output wires are busy; in this case, the box is moved to the end of the scheduling queue.

6.3.4 On region safety and copying

The region memory management we have introduced is particularly simple. Regions in Core Hume are primarily an implementation mechanism for predictable reuse of heap space at the coordination level. In particular, the expression layer has no explicit control over regions (i.e. they cannot be abstracted, applied to functions or even bound to names).

However, the use of regions induces some restrictions in the admissibility of Core Hume programs. Since the input and output wire regions are distinct, expressions used in boxes must construct separate heap results; sharing parts of the input would violate the requirement that heap objects in a region are self-contained.

Example 6.5 The following box does *not* copy a list of integers because the input and output wires reside in different regions.

```
box wrong_copy (xs::[Int]) (ys::[Int])
match
 (xs) -> (xs)     -- wrong: region violation!
;
```

Following the operational semantics to reduce xs in the right-hand side, we verify that rule (6.41) does not apply[3] and relation $\curvearrowright$ is "stuck". □

We call such scheduler configurations *region violations* and say that programs that do not lead to such configurations are *region safe*.

Example 6.6 The region-safe solution to the copy box is to explicitly copy the list using a recursive function.

```
box copy (xs::[Int]) (ys::[Int])
match
 (xs) -> (copyList xs)   -- correct
;

copyList :: [Int] -> [Int];
copyList [] = [];
copyList (x:xs) = x : copy_List xs;
```

More generally, we might need an explicit copy for every heap allocated input that is used in the right-hand-side of a box rule. □

Requiring explicit copies at *every* box expression may seem an unnecessary burden on the programmer. The alternative followed in the prototype Hume compiler and abstract machine (Hammond 2003) is to implement a polymorphic copying operation as part of the runtime system using a bounded-space copying algorithm (Reingold 1973, Jones and Lins 1996). Such approach, however, makes cost analysis at the source-level more difficult because the costs of copying are not expressed in the program: although space costs of a pointer-reversal copying algorithm are bounded, the amount of heap copied and the associated time costs depend not just on the sizes of objects but also on the amount of sharing in the heap. A purely denotational size analysis would not give accurate bounds for these costs.

From a methodological perspective, copying an arbitrary data type is too complex to be considered a single-step operation, particularly if we ultimately aim to reason about *time* as well as *space* costs. Therefore, we argue that it is best to break up

[3] Because one of the side conditions u_i **at** r fails.

copying into smaller, bounded-cost operations; the Core Hume language is already expressive enough for this purpose. Furthermore, since copy operations are simply functions in the core language, we will be able to obtain the costs for copying using the type and effect analysis of Section 6.4, without the need for any extra machinery.

Finally, and although we do not do it here, we believe that the copying functions could, in principle, be generated automatically by the compiler, e.g. in a type-directed way. This would also open up the possibility of employing optimisations to avoid copying when the results are disjoint from the inputs. The advantage of the language-based approach is the ability of expressing this optimisation as a source-to-source transformation.

In summary: we argue that for obtaining static guarantees of bounded space and time, the benefits of explicit copying outweigh the disadvantages.

6.3.5 On bounding costs for complete programs

We have stated informally that time and space cost for a complete Hume program are bounded provided that the costs for the expressions used in boxes and wires are bounded. We now justify why our choice of formal semantics allows us to sustain this claim with regard to stack and heap costs:

- each wire is a bounded communication buffer (in fact, a buffer of size one) between a *single producer* box and a *finite number of consumer* boxes;
- wires are single-buffered (i.e. writing a value for the next iteration is blocked until the current one is consumed); this means that the heap size of wire region is simply the maximum heap used by the corresponding expressions on the right-hand side of box rules;[4]
- the maximum stack required is simply the maximum used in any box or wire expression. To simplify the presentation, we are not considering stack costs for box pattern matching; since patterns are not nested, these costs can be determined trivially (e.g. from the maximum number of bound variables in the box rules);
- finally, the costs associated with the scheduler itself (i.e. the sizes of wire value map, region maps and scheduling queue) are bounded trivially by the total number of boxes and wires.

[4] More precisely, the maximum heap required for the wire regions is *twice* the maximum amount used by the corresponding box expressions (for the two regions associated with consecutive generations).

6.4 A type and effect system for expression costs

In this section we extend the sized type system of Chapter 5 with *effects* that track stack and heap costs. We develop this type and effect analysis in two steps. We start by presenting a straightforward extension of the sized type system that associates the exact stack and heap costs with each syntax node, following the cost model of Section 6.2. This analysis is theoretically sound and as accurate as possible within our framework. However, from a practical point-of-view, it will yield large constraints and can therefore be computationally very expensive.

In a remaining of the section, we refine the analysis as follows. First, we extend Core Hume expressions with *cost annotations*. Second, we define the analysis for the instrumented language. Third, we define a "cost lifting" transformation for annotated programs that does not decrease the overall cost. By employing cost transformations before the analysis we are able to trade a (potential) loss of precision for shorter analysis times.

6.4.1 Latent costs

Following the type and effect discipline, we extend the sized types Section 5.2.2 with "latent effects" on arrows (Talpin and Jouvelot 1994, Reistad and Gifford 1994, Nielson et al. 1999).

$$
\begin{aligned}
\tau &::= \ \cdots && \text{(see Table 5.1)}\\
\nu &::= \ \tau \ \mid \ \vec{\tau} \xrightarrow{s;h} \tau' && \text{(where } s, h \in \mathbf{ZVar}\text{)}\\
\sigma &::= \ \nu \ \mid \ \forall \alpha.\, \sigma\\
\eta &::= \ \langle \sigma,\, \phi \rangle \ \mid \ \forall \ell.\, \eta
\end{aligned}
$$

The annotations s, h are *latent costs* (Reistad and Gifford 1994, Portillo et al. 2003, Vasconcelos and Hammond 2004) representing the stack and heap costs of the function. Like the size annotations of Chapter 5, the latent costs are variables; the actual cost information is expressed separately by a size constraint ϕ. This allows us to express a fully relational analysis where costs can depend on any combination of sizes.

We allow an abbreviation of omitting latent costs on functional types whenever these are free variables, writing simply $\vec{\tau} \to \tau'$.

6.4.2 Maximum terms in constraints

All the stack costs of Table 6.2 involve the maximum of the stack costs for sub-expressions. We can express the maximum with our Presburger-like size constraints using the following equivalence:

$$\ell = \max(s_1,\, s_2) \iff (s_2 \leq s_1 \wedge \ell = s_1) \vee (s_1 \leq s_2 \wedge \ell = s_2) \tag{6.54}$$

However, when generalising the above to n expressions, this translation yields a constraint of size $O(n^2)$. Furthermore, since we are combining stack costs, we are primarily interested in upper-bounds rather than precise ranges. We will therefore allow some imprecision and consider the following abbreviation instead:

$$\ell = s_1 \vee \ell = s_2 \tag{6.55}$$

Clearly (6.54) entails (6.55), so this approximation is sound. The advantage of encoding the maximum as (6.55) is that it yields constraints of size $O(n)$ when generalising to n expressions, i.e.

$$\bigvee_{i=1}^{n} \ell = s_i \tag{6.56}$$

6.4.3 Type and effect judgements

Tables 6.5 and 6.6 present the cost analysis for expressions and declarations as judgements

$$\Gamma \vdash_{\mathsf{COST}} e : \tau \ \$\ s \,; h \mid \phi$$

where τ is a sized type, ϕ is a size constraint and s, h are *effects* delimiting, respectively, the stack and heap costs of executing e. Effects are simple variables; all cost information is expressed together with the size information through the constraint ϕ. Rules in Tables 6.5 and 6.6 are, for the most part, straightforward extensions of the rules of Tables 5.4 and 5.5 with the costs of Table 6.2.

- Integers and variables incur no heap cost and a single unit of stack cost.
- Rule [*Tuple*] illustrates the distinct aggregation of stack and heap costs: the heap cost for a tuple is the sum of the sub-expression heap costs, while the stack cost is the maximum of the sub-expression stack costs.
- Rule [*FunAp*] combines the latent costs of the function with the costs of the argument tuple. Note that, since the language is first-order, the type judgement for the function does not carry stack and heap costs (i.e., function dispatch is done statically).

$$\boxed{\Gamma \vdash_{\mathsf{COST}} e : \tau \ \$\ s \,;h \mid \phi}$$

$$[\mathit{Int}] \quad \Gamma \vdash_{\mathsf{COST}} n : \mathsf{Int}^{\ell} \ \$\ s \,;h \mid \ell = n \wedge s = 1 \wedge h = 0$$

$$[\mathit{Var}] \quad \frac{\Gamma \vdash_{\mathsf{INST}} x : \langle \tau,\, \phi \rangle}{\Gamma \vdash_{\mathsf{COST}} x : \tau \ \$\ s \,;h \mid \phi \wedge s = 1 \wedge h = 0} \quad \mathrm{FZV}(\tau) \cap \mathrm{FZV}(\Gamma) = \emptyset$$

$$[\mathit{FunAp}] \quad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{INST}} f : \langle (\tau_1, \ldots, \tau_n) \xrightarrow{s_1;h_1} \tau,\, \phi_1 \rangle \\ \Gamma \vdash_{\mathsf{COST}} (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n) \ \$\ s_2 \,;h_2 \mid \phi_2\end{array}}{\begin{array}{r}\Gamma \vdash_{\mathsf{COST}} f\ (e_1, \ldots, e_n) : \tau \ \$\ s \,;h \mid \phi_1 \wedge \phi_2 \wedge h = h_1 + h_2 \\ \wedge\, s = \max(1 + n + s_1,\, 1 + s_2)\end{array}}$$

$$[\mathit{ConsAp}] \quad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{INST}} c : \langle (\tau_1, \ldots, \tau_n) \rightarrow \tau,\, \phi_1 \rangle \\ \Gamma \vdash_{\mathsf{COST}} (e_1, \ldots\ e_n) : (\tau_1, \ldots, \tau_n) \ \$\ s \,;h' \mid \phi_2\end{array}}{\Gamma \vdash_{\mathsf{COST}} c\ (e_1, \ldots, e_n) : \tau \ \$\ s \,;h \mid \phi_1 \wedge \phi_2 \wedge h = h' + 1 + n}$$

$$[\mathit{Let}] \quad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{COST}} e_1 : \tau_1 \ \$\ s_1 \,;h_1 \mid \phi_1 \\ x : \tau_1, \Gamma \vdash_{\mathsf{COST}} e_2 : \tau_2 \ \$\ s_2 \,;h_2 \mid \phi_2\end{array}}{\begin{array}{r}\Gamma \vdash_{\mathsf{COST}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \ \$\ s \,;h \mid \phi_1 \wedge \phi_2 \wedge s = \max(s_1,\, 1 + s_2) \\ \wedge\, h = h_1 + h_2\end{array}}$$

$$[\mathit{Case}] \quad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{COST}} e_0 : \tau' \ \$\ s_0 \,;h_0 \mid \phi_0 \qquad \Gamma \vdash_{\mathsf{INST}} c_i : \langle \vec{\tau}''_i \rightarrow \tau',\, \phi'_i \rangle \\ \vec{x}_i : \vec{\tau}''_i, \Gamma \vdash_{\mathsf{COST}} e_i : \tau \ \$\ s_i \,;h_i \mid \phi_i\end{array} \quad (\forall i)}{\begin{array}{l}\Gamma \vdash_{\mathsf{COST}} \mathsf{case}\ e_0\ \mathsf{of}\ \{c_i\ \vec{x}_i \rightarrow e_i\}_{i=1}^{n} : \tau \ \$\ s \,;h \mid \\ \qquad \phi_0 \wedge \bigvee_{i=1}^{n} (\phi_i \wedge \phi'_i \wedge s = \max(s_0,\, s_i + |\vec{x}_i|) \wedge h = h_0 + h_i)\end{array}}$$

$$[\mathit{Tuple}] \quad \frac{\Gamma \vdash_{\mathsf{COST}} e_i : \tau_i \ \$\ s_i \,;h_i \mid \phi_i \qquad (\forall i)}{\begin{array}{r}\Gamma \vdash_{\mathsf{COST}} (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n) \ \$\ s \,;h \mid \bigwedge_{i=1}^{n} \phi_i \wedge h = \sum_{i=1}^{n} h_i \\ \wedge\, s = \max\{n - i + s_i\}_{i=1}^{n}\end{array}}$$

$$[\mathit{Unsize}] \quad \frac{\Gamma \vdash_{\mathsf{COST}} e : \tau \ \$\ s \,;h \mid \phi}{\Gamma \vdash_{\mathsf{COST}} e : [\ell \mapsto \boldsymbol{\omega}]\, \tau \ \$\ s \,;h \mid \phi} \quad \ell \notin \mathrm{FZV}(\Gamma)$$

$$[\mathit{Weaken}] \quad \frac{\Gamma \vdash_{\mathsf{COST}} e : \tau \ \$\ s \,;h \mid \phi}{\Gamma \vdash_{\mathsf{COST}} e : \tau \ \$\ s \,;h \mid \phi'} \quad \phi \vDash \phi'$$

Table 6.5: Cost analysis rules for expressions.

$$\boxed{\Gamma \vdash_{\textsf{COST}} \mathit{decl} : \eta}$$

$$[\mathit{Fun}] \quad \frac{x_1 : \tau_1,\, \ldots,\, x_k : \tau_k,\, \Gamma \vdash_{\textsf{COST}} e : \tau_{k+1} \;\$\; s\,;h \mid \phi}{\Gamma \vdash_{\textsf{COST}} \textsf{let}\ f\ (x_1, \ldots, x_k) = e\ :\ \forall \vec{\ell}.\, \langle \forall \vec{\alpha}.\, (\tau_1, \ldots, \tau_k) \xrightarrow{s;h} \tau_{k+1},\, \phi \rangle}$$

$$[\mathit{Rec}] \quad \frac{\begin{array}{c} f : \forall \vec{\ell}.\, \langle \tau_1 \times \cdots \times \tau_k \xrightarrow{s;h} \tau_{k+1},\, \phi \rangle, \\ x_1 : \tau_1,\, \ldots,\, x_k : \tau_k,\, \Gamma \vdash_{\textsf{COST}} e : \tau_{k+1} \;\$\; s\,;h \mid \phi \end{array}}{\Gamma \vdash_{\textsf{COST}} \textsf{letrec}\ f\ (x_1, \ldots, x_k) = e\ :\ \forall \vec{\ell}.\, \langle \forall \vec{\alpha}.\, (\tau_1, \ldots, \tau_k) \xrightarrow{s;h} \tau_{k+1},\, \phi \rangle}$$

$$\text{where} \quad \begin{array}{ll} \vec{\alpha} & = \bigcup_{i=1}^{k+1} \mathrm{FTV}(\tau_i) \\ \vec{\ell} & = \bigcup_{i=1}^{k+1} \mathrm{FZV}(\tau_i) \cup \mathrm{FZV}(\phi) \cup \{s,\, h\} \end{array}$$

Table 6.6: Cost analysis rules for function declarations.

- Rule [*Case*] expresses the cost of the case as the disjunction of the costs of alternatives. The heap cost for the discriminated expression is always added because the case is strict in that expression.

- Rule [*Weaken*] allows weakening of size and cost information. Since the constraint ϕ captures both size and cost information, there is no need for a separate sub-effecting rule for costs.

- Rules [*Fun*] and [*Rec*] transpose stack and heap costs of executing the function body to the latent costs on the function type.

6.4.4 Extending Core Hume with cost annotations

The rules of Table 6.5 introduce maximum constraints $\ell = \max\{s_i\}_{i=1}^{n}$ for combining the stack costs of any expression with n sub-expressions. Since such a constraint is just an abbreviation for n disjunctions, and the intermediate constraint must be reduced to disjunctive normal form for computing fixed point approximations (see Section 5.5), this leads to an exponential growth of the constraint size with respect to the length of the program. Moreover, the base of this exponential is the number of nodes in the abstract syntax tree rather than (say) the number of execution paths (i.e. nested case alternatives). This compromises the applicability of the analysis formulated in Table 6.5 to large programs.

To improve the scalability of the analysis, we will decouple the type and effect rules from the specific cost model, i.e. the concrete costs derived from the opera-

$$\mathcal{C}_E : \textbf{Expr} \rightarrow \widehat{\textbf{Expr}}$$

$$\mathcal{C}_E[\![\, n \,]\!] \overset{\text{def}}{=} \mathsf{stack}^1\ n$$

$$\mathcal{C}_E[\![\, x \,]\!] \overset{\text{def}}{=} \mathsf{stack}^1\ x$$

$$\mathcal{C}_E[\![f\ (e_1,\ \ldots,\ e_n)]\!] \overset{\text{def}}{=} \mathsf{stack}^{1+n}\ (f\ (\mathsf{stack}^{-1}\, \mathcal{C}_E[\![e_1]\!],\ \ldots,\ \mathsf{stack}^{-n}\, \mathcal{C}_E[\![e_n]\!]))$$

$$\mathcal{C}_E[\![\, c\ () \,]\!] \overset{\text{def}}{=} \mathsf{heap}^1\, \mathsf{stack}^1\ (c\ ())$$

$$\mathcal{C}_E[\![\, c\ (e_1,\ \ldots,\ e_n) \,]\!] \overset{\text{def}}{=} \mathsf{heap}^{1+n}\ (c\ (\mathsf{stack}^{n-1}\, \mathcal{C}_E[\![e_1]\!],\ \ldots,\ \mathsf{stack}^{0}\, \mathcal{C}_E[\![e_n]\!])) \qquad (n \geq 1)$$

$$\mathcal{C}_E[\![\, p\ (e_1,\ \ldots,\ e_n) \,]\!] \overset{\text{def}}{=} \mathsf{heap}^{\gamma}\ (p\ (\mathsf{stack}^{n-1}\, \mathcal{C}_E[\![e_1]\!],\ \ldots,\ \mathsf{stack}^{0}\, \mathcal{C}_E[\![e_n]\!]))$$

$$(\gamma \text{ is the heap cost of primitive } p)$$

$$\mathcal{C}_E[\![\, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \,]\!] \overset{\text{def}}{=} \mathsf{let}\ x = \mathcal{C}_E[\![e_1]\!]\ \mathsf{in}\ \mathsf{stack}^1\, \mathcal{C}_E[\![e_2]\!]$$

$$\mathcal{C}_E \left[\!\!\left[\begin{array}{l} \mathsf{case}\ e_0\ \mathsf{of} \\ \{c_i\ \vec{x}_i \rightarrow e_i\}_{i=1}^{n} \end{array} \right]\!\!\right] \overset{\text{def}}{=} \mathsf{case}\ \mathcal{C}_E[\![e_0]\!]\ \mathsf{of}\ \{c_i\ \vec{x}_i \rightarrow \mathsf{stack}^{|\vec{x}_i|}\, \mathcal{C}_E[\![e_i]\!]\,\}_{i=1}^{n}$$

$$\mathcal{C}_D : \textbf{Decl} \rightarrow \widehat{\textbf{Decl}}$$

$$\mathcal{C}_D[\![\mathsf{let(rec)}\ f\ \vec{x} = e]\!] \overset{\text{def}}{=} \mathsf{let(rec)}\ f\ \vec{x} = \mathcal{C}_E[\![e]\!]$$

Table 6.7: Cost annotations for the Core Hume abstract machine.

tional semantics of Section 6.2. For that purpose, we introduce a syntactic class of *annotated expressions* $\widehat{e}$ that extends the core language of Table 4.1 with explicit annotations for costs.

$$\begin{array}{rcll} \widehat{e} & ::= & \cdots & \text{(see Table 4.1)} \\ & \mid & \mathsf{stack}^k\ \widehat{e} & k \text{ units of stack cost} \\ & \mid & \mathsf{heap}^k\ \widehat{e} & k \text{ units of heap cost} \end{array}$$

Cost annotations associate costs with expressions: the annotation $\mathsf{stack}^k\ \widehat{e}$ (respectively, $\mathsf{heap}^k\ \widehat{e}$) assigns k integer units of stack cost (respectively, heap cost) to an expression $\widehat{e}$; note that $\widehat{e}$ can contain nested cost annotations.

The stack and heap annotations mark the basic costs in the underlying cost model, while the type and effect rules specify how to combine the costs from subexpression for each syntax node. Of course, this decoupling of the type rules from the specific cost model means that we must start from a suitably annotated source program. Table 6.7 defines two annotation functions $\mathcal{C}_E$ and $\mathcal{C}_D$ for the stack and heap costs of the Core Hume region machine of Section 6.2. The cost annotations are derived from the big-step semantics with relative stack and heap costs (Table 6.2).

Example 6.7 Consider the list append function

$$\begin{aligned} \mathsf{letrec}\ app\ (xs, ys) = \mathsf{case}\ xs\ \mathsf{of} \\ \mathsf{Nil} \to ys \\ \mid \mathsf{Cons}\ (x, xs') \to \mathsf{Cons}\ (x, app\ (xs', ys)) \end{aligned} \tag{6.57}$$

Annotating (6.57) following the definitions of Table 6.7 yields:

$$\begin{aligned} & \mathsf{letrec}\ app\ (xs, ys) \\ & \quad = \mathsf{case}\ (\mathsf{stack}^1\ xs)\ \mathsf{of} \\ & \qquad \mathsf{Nil} \to \mathsf{stack}^1\ ys \\ & \quad \mid \mathsf{Cons}\ (x, xs') \to \mathsf{stack}^2 \mathsf{heap}^3\ \mathsf{Cons}\ (\mathsf{stack}^1 \mathsf{stack}^1\ x, \\ & \qquad \mathsf{stack}^0 \mathsf{stack}^3\ app\ (\mathsf{stack}^{-1} \mathsf{stack}^1\ xs', \mathsf{stack}^{-2} \mathsf{stack}^1\ ys)) \end{aligned} \tag{6.58}$$

In the annotated expression (6.58) syntax nodes that incur stack pushes or heap allocation are explicit. For example, we can see that heap allocation is only performed in the the Cons alternative (because there is no heap annotation in the Nil alternative). □

6.4.5 Cost analysis for annotated expressions

The type and effect rules for cost-annotated expressions are presented in Tables 6.8 and 6.9. Rules for declarations remain unchanged (except for the use of cost-

$$\boxed{\Gamma \vdash_{\textsf{COST}} \widehat{e} : \tau \ \$ \ s\,;h \mid \phi}$$

$$[\mathit{Int}] \qquad \Gamma \vdash_{\textsf{COST}} n : \mathsf{Int}^{\ell} \ \$ \ s\,;h \mid \ell = n \wedge s = 0 \wedge h = 0$$

$$[\mathit{Var}] \qquad \frac{\Gamma \vdash_{\textsf{INST}} x : \langle \tau,\, \phi \rangle}{\Gamma \vdash_{\textsf{COST}} x : \tau \ \$ \ s\,;h \mid \phi \wedge s = 0 \wedge h = 0} \qquad \mathrm{FZV}(\tau) \cap \mathrm{FZV}(\Gamma) = \emptyset$$

$$[\mathit{Tuple}] \qquad \frac{\Gamma \vdash_{\textsf{COST}} \widehat{e}_i : \tau_i \ \$ \ s_i\,;h_i \mid \phi_i \qquad (\forall i)}{\begin{array}{r}\Gamma \vdash_{\textsf{COST}} (\widehat{e}_1, \ldots, \widehat{e}_n) : (\tau_1, \ldots, \tau_n) \ \$ \ s\,;h \mid \ \bigwedge_{i=1}^{n} \phi_i \wedge s = \max\{s_i\}_{i=1}^{n} \\ \wedge\, h = \sum_{i=1}^{n} h_i\end{array}}$$

$$[\mathit{FunAp}] \qquad \frac{\begin{array}{c}\Gamma \vdash_{\textsf{INST}} f : \langle (\tau_1, \ldots, \tau_n) \xrightarrow{s_1;h_1} \tau,\, \phi_1 \rangle \\ \Gamma \vdash_{\textsf{COST}} (\widehat{e}_1, \ldots, \widehat{e}_n) : (\tau_1, \ldots, \tau_n) \ \$ \ s_2\,;h_2 \mid \phi_2\end{array}}{\begin{array}{r}\Gamma \vdash_{\textsf{COST}} f\ (\widehat{e}_1, \ldots, \widehat{e}_n) : \tau \ \$ \ s\,;h \mid \phi_1 \wedge \phi_2 \wedge s = \max(s_1,\, s_2) \\ \wedge\, h = h_1 + h_2\end{array}}$$

$$[\mathit{ConsAp}] \qquad \frac{\begin{array}{c}\Gamma \vdash_{\textsf{INST}} c : \langle (\tau_1, \ldots, \tau_n) \rightarrow \tau,\, \phi_1 \rangle \\ \Gamma \vdash_{\textsf{COST}} (\widehat{e}_1, \ldots, \widehat{e}_n) : (\tau_1, \ldots, \tau_n) \ \$ \ s\,;h \mid \phi_2\end{array}}{\Gamma \vdash_{\textsf{COST}} c\ (\widehat{e}_1, \ldots, \widehat{e}_n) : \tau \ \$ \ s\,;h \mid \phi_1 \wedge \phi_2}$$

$$[\mathit{Heap}] \qquad \frac{\Gamma \vdash_{\textsf{COST}} \widehat{e} : \tau \ \$ \ s\,;h' \mid \phi}{\Gamma \vdash_{\textsf{COST}} \mathsf{heap}^{k}\ \widehat{e} : \tau \ \$ \ s\,;h \mid \phi \wedge h = h' + k}$$

$$[\mathit{Stack}] \qquad \frac{\Gamma \vdash_{\textsf{COST}} \widehat{e} : \tau \ \$ \ s'\,;h \mid \phi}{\Gamma \vdash_{\textsf{COST}} \mathsf{stack}^{k}\ \widehat{e} : \tau \ \$ \ s\,;h \mid \phi \wedge s = s' + k}$$

$$[\mathit{Unsize}] \qquad \frac{\Gamma \vdash_{\textsf{COST}} \widehat{e} : \tau \ \$ \ s\,;h \mid \phi}{\Gamma \vdash_{\textsf{COST}} \widehat{e} : [\ell \mapsto \boldsymbol{\omega}]\, \tau \ \$ \ s\,;h \mid \phi} \qquad \ell \notin \mathrm{FZV}(\Gamma)$$

$$[\mathit{Weaken}] \qquad \frac{\Gamma \vdash_{\textsf{COST}} \widehat{e} : \tau \ \$ \ s\,;h \mid \phi}{\Gamma \vdash_{\textsf{COST}} \widehat{e} : \tau \ \$ \ s\,;h \mid \phi'} \qquad \phi \vDash \phi'$$

Table 6.8: Cost analysis rules for annotated expressions.

$$[Let]\quad \frac{\Gamma \vdash_{\mathsf{COST}} \widehat{e}_1 : \tau_1 \mathbin{\$} s_1 \,;h_1 \mid \phi_1 \qquad x:\tau_1,\, \Gamma \vdash_{\mathsf{COST}} \widehat{e}_2 : \tau_2 \mathbin{\$} s_2 \,;h_2 \mid \phi_2}{\Gamma \vdash_{\mathsf{COST}} \mathsf{let}\ x = \widehat{e}_1\ \mathsf{in}\ \widehat{e}_2 : \tau_2 \mathbin{\$} s\,;h \mid \phi_1 \wedge \phi_2 \wedge s = \max(s_1,\, s_2) \wedge h = h_1 + h_2}$$

$$[Case]\quad \frac{\Gamma \vdash_{\mathsf{COST}} \widehat{e}_0 : \tau' \mathbin{\$} s_0\,;h_0 \mid \phi_0 \qquad \Gamma \vdash_{\mathsf{INST}} c_i : \langle \vec{\tau}''_i \to \tau',\, \phi'_i\rangle \qquad \vec{x}_i : \vec{\tau}''_i,\, \Gamma \vdash_{\mathsf{COST}} \widehat{e}_i : \tau \mathbin{\$} s_i\,;h_i \mid \phi_i \quad (\forall i)}{\Gamma \vdash_{\mathsf{COST}} \mathsf{case}\ \widehat{e}_0\ \mathsf{of}\ \{c_i\ \vec{x}_i \to \widehat{e}_i\}_{i=1}^n : \tau \mathbin{\$} s\,;h \mid \phi_0 \wedge \bigvee_{i=1}^n (\phi_i \wedge \phi'_i \wedge s = \max(s_0,\, s_i) \wedge h = h_0 + h_i)}$$

Table 6.9: Cost analysis rules for annotated expressions (continued).

annotated expressions).

Two new rules [*Heap*] and [*Stack*] are introduced to add heap and stack costs, respectively. Note that these are the only rules that add positive costs—all other rules simply combine the costs of sub-expressions.

Introducing explicit cost annotations allows us to decouple the analysis from the particular cost model of our implementation by simply modifying the annotation phase (i.e. functions $\mathcal{C}_E$ and $\mathcal{C}_D$). That is, we get a parametric analysis "for free".

Cost annotations also open up the possibility of transforming the annotated program before performing the cost analysis. In particular, we will employ transformations that do not reduce costs (and are therefore sound) but reduce the complexity of synthesised constraints by "lifting" costs from child to parent nodes in the syntax tree. For example, starting from the annotated expression

$$f\ (\mathsf{stack}^2\ \mathsf{heap}^1\ e_1,\, \mathsf{stack}^1\ \mathsf{heap}^1\ e_2)$$

we can the lift cost annotations one level by taking the *maximum* of stack and the *sum* of the heap costs:

$$\mathsf{stack}^2\ \mathsf{heap}^2\ f\ (e_1, e_2) \tag{6.59}$$

Cost analysis of expression (6.59) will not generate a maximum constraint for the stack cost of the arguments because e_1, e_2 have no stack annotations. Of course, cost lifting might increase the overall predicted costs, so this transformation trades precision for efficiency.

$$\mathsf{heap}^{\gamma_1}\ (\mathsf{heap}^{\gamma_2}\ \widehat{e}) \rightsquigarrow \mathsf{heap}^{\gamma_1+\gamma_2}\ \widehat{e} \tag{6.60}$$

$$\mathsf{stack}^{\delta_1}\ (\mathsf{stack}^{\delta_2}\ \widehat{e}) \rightsquigarrow \mathsf{stack}^{\delta_1+\delta_2}\ \widehat{e} \tag{6.61}$$

$$\mathsf{heap}^{\gamma}\ (\mathsf{stack}^{\delta}\ \widehat{e}) \rightsquigarrow \mathsf{stack}^{\delta}\ (\mathsf{heap}^{\gamma}\ \widehat{e}) \tag{6.62}$$

$$\mathsf{stack}^{\delta}\ (\mathsf{heap}^{\gamma}\ \widehat{e}) \rightsquigarrow \mathsf{heap}^{\gamma}\ (\mathsf{stack}^{\delta}\ \widehat{e}) \tag{6.63}$$

$$\begin{aligned} f\ (\mathsf{heap}^{\gamma_1}\ \widehat{e}_1, \ldots, \mathsf{heap}^{\gamma_n}\ \widehat{e}_n) &\rightsquigarrow \mathsf{heap}^{\gamma}\ f\ (\widehat{e}_1, \ldots, \widehat{e}_n)\,, \\ c\ (\mathsf{heap}^{\gamma_1}\ \widehat{e}_1, \ldots, \mathsf{heap}^{\gamma_n}\ \widehat{e}_n) &\rightsquigarrow \mathsf{heap}^{\gamma}\ c\ (\widehat{e}_1, \ldots, \widehat{e}_n)\,, \\ p\ (\mathsf{heap}^{\gamma_1}\ \widehat{e}_1, \ldots, \mathsf{heap}^{\gamma_n}\ \widehat{e}_n) &\rightsquigarrow \mathsf{heap}^{\gamma}\ p\ (\widehat{e}_1, \ldots, \widehat{e}_n)\,, \\ &\qquad \text{where } \gamma = \sum_{i=1}^{n} \gamma_i \end{aligned} \tag{6.64}$$

$$\mathsf{let}\ x = (\mathsf{heap}^{\gamma_1}\ \widehat{e}_1)\ \mathsf{in}\ (\mathsf{heap}^{\gamma_2}\ \widehat{e}_2) \rightsquigarrow \mathsf{heap}^{\gamma_1+\gamma_2}\ (\mathsf{let}\ x = \widehat{e}_1\ \mathsf{in}\ \widehat{e}_2) \tag{6.65}$$

$$\begin{aligned} &\mathsf{case}\ (\mathsf{heap}^{\gamma_0}\ \widehat{e}_0)\ \mathsf{of}\ \{c_i\ xs_i \to \mathsf{heap}^{\gamma_i}\ \widehat{e}_i\}_{i=1}^{n} \rightsquigarrow \\ &\qquad\qquad \mathsf{heap}^{\gamma}\ (\mathsf{case}\ \widehat{e}_0\ \mathsf{of}\ \{c_i\ xs_i \to \widehat{e}_i\}_{i=1}^{n})\,, \\ &\qquad\qquad \text{where } \gamma = \gamma_0 + \max_{i=1}^{n} \gamma_i \end{aligned} \tag{6.66}$$

$$\begin{aligned} f\ (\mathsf{stack}^{\delta_1}\ \widehat{e}_1, \ldots, \mathsf{stack}^{\delta_n}\ \widehat{e}_n) &\rightsquigarrow \mathsf{stack}^{\delta}\ f\ (\widehat{e}_1, \ldots, \widehat{e}_n)\,, \\ c\ (\mathsf{stack}^{\delta_1}\ \widehat{e}_1, \ldots, \mathsf{stack}^{\delta_n}\ \widehat{e}_n) &\rightsquigarrow \mathsf{stack}^{\delta}\ c\ (\widehat{e}_1, \ldots, \widehat{e}_n)\,, \\ p\ (\mathsf{stack}^{\delta_1}\ \widehat{e}_1, \ldots, \mathsf{stack}^{\delta_n}\ \widehat{e}_n) &\rightsquigarrow \mathsf{stack}^{\delta}\ p\ (\widehat{e}_1, \ldots, \widehat{e}_n)\,, \\ &\qquad \text{where } \delta = \max_{i=1}^{n} \delta_i \end{aligned} \tag{6.67}$$

$$\mathsf{let}\ x = (\mathsf{stack}^{\delta_1}\ \widehat{e}_1)\ \mathsf{in}\ (\mathsf{stack}^{\delta_2}\ \widehat{e}_2) \rightsquigarrow \mathsf{stack}^{\max(\delta_1,\delta_2)}\ (\mathsf{let}\ x = \widehat{e}_1\ \mathsf{in}\ \widehat{e}_2) \tag{6.68}$$

$$\begin{aligned} &\mathsf{case}\ (\mathsf{stack}^{\delta_0}\ \widehat{e}_0)\ \mathsf{of}\ \{c_i\ xs_i \to \mathsf{stack}^{\delta_i}\ \widehat{e}_i\}_{i=1}^{n} \rightsquigarrow \\ &\qquad\qquad \mathsf{stack}^{\delta}\ (\mathsf{case}\ \widehat{e}_0\ \mathsf{of}\ \{c_i\ xs_i \to \widehat{e}_i\}_{i=1}^{n})\,, \\ &\qquad\qquad \text{where } \delta = \max_{i=0}^{n} \delta_i \end{aligned} \tag{6.69}$$

Table 6.10: Cost-lifting transformation for annotated expressions.

6.4.6 Cost-lifting transformations

The cost annotation functions of Table 6.7 specify the most accurate description of the costs of Table 6.2. For example, annotating a let-expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ yields:

$$\mathsf{let}\ x = \mathcal{C}_E[\![e_1]\!]\ \mathsf{in}\ \mathsf{stack}^1\,\mathcal{C}_E[\![e_2]\!]$$

The stack annotation reflects the operational semantics cost of one stack position for the bound variable. Since e_2 is evaluated with one extra entry on the stack relative to e_1, this cost is attached only to e_2.

It would also be sound (though possibly less precise) to attach the stack cost to the let-expression as a whole:

$$\mathsf{stack}^1\ (\mathsf{let}\ x = \mathcal{C}_E[\![e_1]\!]\ \mathsf{in}\ \mathcal{C}_E[\![e_2]\!])$$

Such "lifting" of costs from inner to outer syntax nodes can be performed systematically. Table 6.10 defines this transformation as a rewrite relation $\rightsquigarrow$ between annotated expressions. Informally, when the relation $\widehat{e} \rightsquigarrow \widehat{e}'$ holds then $\widehat{e}$ and $\widehat{e}'$ yield the same value and the costs of $\widehat{e}'$ are an upper-bound of those for $\widehat{e}$ (a formal statement of this property will be presented in Section 6.5.2). Therefore, it is safe to systematically apply the rules of Table 6.10 to lift costs from sub-expressions to enclosing ones and perform the cost analysis on the resulting expression.

Example 6.8 Applying the transformations of Table 6.10 to the annotated append (6.58), we can systematically lift costs from the innermost expressions to the outermost ones to obtain:

$$\begin{aligned}\mathsf{letrec}\ \mathit{app}\ (xs, ys) = {} & \mathsf{stack}^5\,\mathsf{heap}^3\,(\mathsf{case}\ xs\ \mathsf{of}\\ & \mathsf{Nil} \rightarrow ys\\ & \mid \mathsf{Cons}\ (x, xs') \rightarrow \mathsf{Cons}\ (x, \mathit{app}\ (xs', ys)))\end{aligned} \tag{6.70}$$

The costs annotated in (6.70) are over-approximated compared to (6.58) and to the underlying cost model (the Core Hume machine). For example, the heap cost is always added regardless of which branch of the case is taken. □

The two annotated versions of append in Examples 6.7 and 6.8 represent two extremes: the version given by the functions $\mathcal{C}_E$ and $\mathcal{C}_D$ yields the most precise cost annotation—essentially the same cost information obtained by the naive type and effect analysis of Tables 6.5 and 6.6. The version resulting from lifting all cost information to the outermost node gives the least precise cost annotation.

In practice, we will use cost lifting transformations to partially simplify cost information. The heuristic employed is to retain cost annotations up-to a specified "depth" and to lift more deeply nested annotations. A depth of infinity gives the highest precision and a depth of zero gives the least precision.

This heuristic allows a gradual loss of accuracy rather than in a single step. Furthermore, the depth parameter can be chosen individually for different functions. We have implemented this heuristic and obtained good results in improving analysis times with little loss of precision (see Section 7.2.10).

6.5 Soundness

In this section we prove the soundness of the type and effect cost analysis. In the interest of generality, we will not formulate soundness with respect to the particular abstract machine of Section 6.2; instead, we will extend the denotational semantics of Section 4.2.3 for cost instrumented expressions and prove the analysis correct with respect to the instrumented semantics.

6.5.1 Cost instrumented semantics

Table 6.11 presents a semantics for core Hume expressions annotated with stack and heap costs. Note that the semantics is still denotational, albeit involving somewhat more complex domains that record costs as well as the results of computations: the denotations of expressions are now elements of $(\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\bot$; the denotations of functions are elements of $[\mathbf{V} \to (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\bot]$. It is straightforward to verify that both domains are still CPOs.

Semantic domains with costs

A *cost-instrumented value* is either $\bot$ or a tuple $\lfloor\langle v, \delta, \gamma\rangle\rfloor$ where $v \in \mathbf{V}$ and δ, γ are natural numbers representing the stack and heap costs for computing v. Similarly, if $f \in [\mathbf{V} \to (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\bot]$ is the cost-instrumented denotation of some function, $v \in \mathbf{V}$ and $f(v) = \lfloor\langle u, \delta, \gamma\rangle\rfloor$, then u is the function result value and δ, γ are the maximum stack and heap costs required in the computation of u.

The cost-instrumented semantics is given by $\mathcal{E}_{\mathsf{C}}$ for expressions and $\mathcal{D}_{\mathsf{C}}$ for functions. Note that environments for functional values are extended with costs but environments for zero-order values are not; this is because the latter represent fully-reduced values, while the former represent computations that can incur latent costs (see also the next section on the relation with monadic semantics).

Only the annotations stack^k and heap^k add positive costs; all other expressions simply combine the costs of sub-expressions using a "cost let" meta-expression; this

$$
\begin{array}{rcl}
\mathbf{Env} & \stackrel{\mathrm{def}}{=} & \mathbf{Var} \to \mathbf{V}_\perp \\
\mathbf{Fenv}_{\mathsf{C}} & \stackrel{\mathrm{def}}{=} & \mathbf{Var} \to [\mathbf{V} \to (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\perp]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_{\mathsf{C}} & : & \mathbf{Expr} \to \mathbf{Fenv}_{\mathsf{C}} \to \mathbf{Env} \to (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\perp \\
\mathcal{E}_{\mathsf{C}}\llbracket\, n \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \lfloor\langle n,\, 0,\, 0\rangle\rfloor \\
\mathcal{E}_{\mathsf{C}}\llbracket\, x \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \text{let } v \Leftarrow \rho(x).\ \lfloor\langle v,\, 0,\, 0\rangle\rfloor \\
\mathcal{E}_{\mathsf{C}}\llbracket\, c\ \vec{e} \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \text{let}_{\mathsf{C}}\ v \Leftarrow \mathcal{E}_{\mathsf{C}}\llbracket\, \vec{e} \,\rrbracket\ \varphi\ \rho.\ \lfloor\langle\langle c,\, v\rangle,\, 0,\, 0\rangle\rfloor \\
\mathcal{E}_{\mathsf{C}}\llbracket\, f\ \vec{e} \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \text{let}_{\mathsf{C}}\ v \Leftarrow \mathcal{E}_{\mathsf{C}}\llbracket\, \vec{e} \,\rrbracket\ \varphi\ \rho.\ \varphi_f\ v \\
\mathcal{E}_{\mathsf{C}}\llbracket\, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \text{let}_{\mathsf{C}}\ v \Leftarrow \mathcal{E}_{\mathsf{C}}\llbracket\, e_1 \,\rrbracket\ \varphi\ \rho.\ \mathcal{E}_{\mathsf{C}}\llbracket\, e_2 \,\rrbracket\ \varphi\ \rho[x \mapsto v] \\
\mathcal{E}_{\mathsf{C}}\llbracket\, \mathsf{case}\ e\ \mathsf{of}\ \mathit{alts} \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \text{let}_{\mathsf{C}}\ v \Leftarrow \mathcal{E}_{\mathsf{C}}\llbracket\, e \,\rrbracket\ \varphi\ \rho.\ \mathcal{A}_{\mathsf{C}}\llbracket\, \mathit{alts} \,\rrbracket\ \varphi\ \rho\ v \\
\mathcal{E}_{\mathsf{C}}\llbracket\, \mathsf{stack}^k\ e \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \text{let } \langle v, s, h\rangle \Leftarrow \mathcal{E}_{\mathsf{C}}\llbracket\, e \,\rrbracket\ \varphi\ \rho.\ \lfloor\langle v, s+k, h\rangle\rfloor \\
\mathcal{E}_{\mathsf{C}}\llbracket\, \mathsf{heap}^k\ e \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \text{let } \langle v, s, h\rangle \Leftarrow \mathcal{E}_{\mathsf{C}}\llbracket\, e \,\rrbracket\ \varphi\ \rho.\ \lfloor\langle v, s, h+k\rangle\rfloor
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_{\mathsf{C}}\llbracket\, () \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \lfloor\langle \mathbf{u},\, 0,\, 0\rangle\rfloor \\
\mathcal{E}_{\mathsf{C}}\llbracket\, (e_1, \ldots, e_k) \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \ \text{let}_{\mathsf{C}}\ v_1 \Leftarrow \mathcal{E}_{\mathsf{C}}\llbracket\, e_1 \,\rrbracket\ \varphi\ \rho. \\
& & \quad\vdots \\
& & \ \text{let}_{\mathsf{C}}\ v_k \Leftarrow \mathcal{E}_{\mathsf{C}}\llbracket\, e_k \,\rrbracket\ \varphi\ \rho.\ \lfloor\langle\langle v_1, \ldots, v_k\rangle,\, 0,\, 0\rangle\rfloor
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{A}_{\mathsf{C}} & : & \mathbf{Alts} \to \mathbf{Fenv}_{\mathsf{C}} \to \mathbf{Env} \to \mathbf{V} \to (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\perp \\
\mathcal{A}_{\mathsf{C}}\llbracket\, \{c_i\ \vec{x}_i \to e_i\}_{i=1}^{k} \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \ \lambda\langle v',\, v\rangle.\ \text{case } v' \text{ of} \\
& & \qquad c_1.\ \mathcal{M}_{\mathsf{C}}\llbracket\, c_1\ \vec{x}_1 \to e_1 \,\rrbracket\ \varphi\ \rho\ v \quad | \\
& & \qquad\ \vdots \\
& & \qquad c_k.\ \mathcal{M}_{\mathsf{C}}\llbracket\, c_k\ \vec{x}_k \to e_k \,\rrbracket\ \varphi\ \rho\ v \\
\mathcal{M}_{\mathsf{C}} & : & \mathbf{Alt} \to \mathbf{Fenv}_{\mathsf{C}} \to \mathbf{Env} \to \mathbf{V} \to (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\perp \\
\mathcal{M}_{\mathsf{C}}\llbracket\, c\ (x_1, \ldots, x_k) \to e \,\rrbracket\ \varphi\ \rho & \stackrel{\mathrm{def}}{=} & \lambda\langle v_1, \ldots, v_k\rangle.\ \mathcal{E}_{\mathsf{C}}\llbracket\, e \,\rrbracket\ \varphi\ \rho[x_1 \mapsto v_1,\ \ldots,\ x_k \mapsto v_k]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{D}_{\mathsf{C}} & : & \mathbf{Decl} \to \mathbf{Fenv}_{\mathsf{C}} \to [\mathbf{V} \to (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\perp] \\
\mathcal{D}_{\mathsf{C}}\llbracket\, \mathsf{let}\ f\ (x_1, \ldots, x_k) = e \,\rrbracket\ \varphi & \stackrel{\mathrm{def}}{=} & \lambda\langle v_1, \ldots, v_k\rangle.\ \mathcal{E}_{\mathsf{C}}\llbracket\, e \,\rrbracket\ \varphi\ \rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] \\
\mathcal{D}_{\mathsf{C}}\llbracket\, \mathsf{letrec}\ f\ (x_1, \ldots, x_k) = e \,\rrbracket\ \varphi & \stackrel{\mathrm{def}}{=} & \mathit{fix}(\mathcal{F}),
\end{array}
$$

$$
\text{where } \mathcal{F} = \lambda F.\ \lambda\langle v_1, \ldots, v_k\rangle.\ \mathcal{E}_{\mathsf{C}}\llbracket\, e \,\rrbracket\ \varphi[f \mapsto F]\ \rho_0[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]
$$

Table 6.11: Cost semantics for Core Hume expressions

is defined in terms of the strict let of Section 4.2.3 (where v_i, δ_i and γ_i are meta-variables for values and costs):

$$\begin{aligned} \mathrm{let}_{\mathsf{C}}\ v_1 \Leftarrow E_1.\ E_2 \overset{\mathrm{def}}{=}\ & \mathrm{let}\ \langle v_1,\ \delta_1,\ \gamma_1\rangle \Leftarrow E_1. \\ & (\mathrm{let}\ \langle v_2,\ \delta_2,\ \gamma_2\rangle \Leftarrow E_2. \\ & \quad \lfloor\langle v_2,\ \max(\delta_1,\delta_2),\ \gamma_1+\gamma_2\rangle\rfloor) \end{aligned} \tag{6.71}$$

Equation (6.71) specifies that the stack cost of $\mathrm{let}_{\mathsf{C}}\ v_1 \Leftarrow E_1.\ E_2$ is the *maximum* of the stack costs of E_1 and E_2, while the heap cost is the *sum* of the heap costs of E_1 and E_2.

Relation with monadic semantics

We remark that the cost instrumented semantics is an instance of a monadic semantics where basic values $\mathbf{V}$ are encapsulated in a monad $\mathsf{C}\,\mathbf{V} = (\mathbf{V}\times\mathbb{N}\times\mathbb{N})_\bot \simeq (\mathbf{V}\times(\mathbb{N}\times\mathbb{N}))_\bot$ for *complexity* and *partiality* (Wadler 1998*a*, Benton et al. 2000):

- $(\mathbb{N}\times\mathbb{N},\ (0,0),\ \star)$ is the *cost monoid* with zero element $(0,0)$ and inner operation $(\delta_1,\gamma_1)\star(\delta_2,\gamma_2) \overset{\mathrm{def}}{=} (\max(\delta_1,\delta_2),\ \gamma_1+\gamma_2)$;
- $\mathrm{let}_{\mathsf{C}}$ is the *monadic bind* operation;
- $\lambda v.\ \lfloor\langle v,\ 0,\ 0\rangle\rfloor$ is the *monadic unit*;
- stack^k and heap^k are the increment operations of the complexity monad.

We can now see that the cost instrumented semantics domain for functions corresponds to the standard call-by-value translation of $\mathbf{V}\to\mathbf{V}$ into the complexity and partiality monad:

$$\begin{aligned} (\mathbf{V}\to\mathbf{V})^{cbv} &= \mathbf{V}^{cbv} \to (\mathsf{C}\ \mathbf{V}^{cbv}) \\ &= \mathbf{V}\to\mathsf{C}\ \mathbf{V} \\ &= \mathbf{V}\to(\mathbf{V}\times\mathbb{N}\times\mathbb{N})_\bot \end{aligned}$$

6.5.2 Soundness of cost lifting

Our first result formalises the notion of soundness for cost lifting, namely, that the transformations of Table 6.10 always yield an annotated program with greater or equal costs than the original. This result is orthogonal to the cost analysis which computes an approximation to the annotated costs; the soundness of the latter will be stated in Section 6.5.3.

Lemma 6.9 (Soundness of cost lifting) *If* $\mathcal{E}_{\mathsf{C}}[\![\, \widehat{e} \,]\!]\, \varphi\, \rho = \lfloor \langle v, \delta, \gamma \rangle \rfloor$ *and* $\widehat{e} \rightsquigarrow \widehat{e}'$*, then* $\mathcal{E}_{\mathsf{C}}[\![\, \widehat{e}' \,]\!]\, \varphi\, \rho = \lfloor \langle v, \delta', \gamma' \rangle \rfloor$ *with* $\delta \leq \delta'$ *and* $\gamma \leq \gamma'$*.*

Proof sketch: Note that that the statement concerns a single rewriting step. We can therefore consider each of the rules in Table 6.10 separately; in each case the result follows directly from the unfolding the definitions of the cost semantics of Table 6.11. □

6.5.3 Soundness of cost analysis for expressions

To formulate and prove the soundness of the cost analysis, we first define a size function augmented with costs: if $v \in (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\perp$ is a cost instrumented value, τ is a sized type and s, h are cost variables, then $\mathcal{S}^{\mathsf{C}}_{\Sigma}(v :: \tau \,!\, s; h)$ is a formula expressing the size and cost of v.

Definition 6.10 (Size and cost function) *Given assumptions* Σ *for data constructors, the semantic size and cost function is:*

$$\mathcal{S}^{\mathsf{C}}_{\Sigma}(\perp :: \tau \,!\, s; h) \overset{def}{=} \mathit{False}$$

$$\mathcal{S}^{\mathsf{C}}_{\Sigma}(\lfloor \langle v, \delta, \gamma \rangle \rfloor :: \tau \,!\, s; h) \overset{def}{=} \mathcal{S}_{\Sigma}(\lfloor v \rfloor :: \tau) \wedge s = \delta \wedge h = \gamma$$

where $\mathcal{S}$ *is defined in Table 5.10 of Section 5.4.*

Using $\mathcal{S}^{\mathsf{C}}$ we can now define a type semantics for first order function types augmented with costs.

Definition 6.11 *The semantics of first order sized types with costs is given by*

$$\begin{aligned} \mathcal{T}^{\mathsf{C}}_{\Sigma}[\![\langle \forall \vec{\alpha}.\, \vec{\tau} \xrightarrow{s;h} \tau',\, \phi \rangle]\!] \overset{def}{=} \{\, F \in [\mathbf{V} \to (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\perp] :& \\ \pi \circ F \in \mathcal{T}[\![\, \forall \vec{\alpha}.\, \vec{\tau} \to \tau' \,]\!] \chi_0 \ \mathit{and}& \\ \forall v \in \mathbf{V} \quad \mathcal{S}_{\Sigma}(\lfloor v \rfloor :: \vec{\tau}) \wedge \mathcal{S}^{\mathsf{C}}_{\Sigma}(F(v) :: \tau' \,!\, s; h) \vDash \phi \,\}& \end{aligned}$$

where χ_0 *is the empty type environment and* $\pi : (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_\perp \to \mathbf{V}_\perp$ *is the lifted first projection.*

Using this type semantics with costs we now extend the soundness precondition that states that a functional environment φ satisfies the type assumptions Γ. Except for the use of $\mathcal{T}^{\mathsf{C}}_{\Sigma}$ instead of $\mathcal{T}_{\Sigma}$, the condition is identical to Definition 5.19:

Definition 6.12 *We say that* φ satisfies Γ *and write* $\varphi \models \Gamma$ *if and only if for all* $f \in \mathrm{dom}(\varphi)$ *we have* $\varphi_f \in \mathcal{T}^{\mathsf{C}}_{\Sigma}[\![\Gamma(f)]\!]$*.*

Note that no analogous extension is needed for simple environments ρ because these are not extended with costs. Therefore, the size-only condition of Definition 5.18 is sufficient.

We can now state the soundness of cost analysis for expressions as an extension of the corresponding result of the size-only analysis (Theorem 5.21). Informally, the following theorem states that the size and cost constraint approximates the cost instrumented semantics.

Theorem 6.13 (Soundness of cost analysis for expressions) *If* Σ *is consistent,* $\rho \models \Gamma$, $\varphi \models \Gamma$ *and* $\Gamma, \Sigma \vdash_{\mathit{COST}} e : \tau \ \$ \ s\,; h \mid \phi$, *then* $\mathcal{S}^{\mathsf{C}}_{\Sigma}(\mathcal{E}_{\mathsf{C}}[\![\, e \,]\!]\, \varphi\, \rho :: \tau\, !\, s; h) \wedge \mathcal{S}_{\Sigma}(\rho :: \Gamma) \vDash \exists X.\, \phi$, *where* $X = \mathrm{FZV}(\phi) \setminus (\mathrm{FZV}(\tau) \cup \mathrm{FZV}(\Gamma) \cup \{s, h\})$.

Proof sketch: The proof is by induction on the derivation $\Gamma, \Sigma \vdash_{\mathsf{COST}} e : \tau \ \$ \ s\,; h \mid \phi$ and follows the structure of the soundness proof for size analysis (Theorem 5.21). □

6.5.4 Soundness of cost analysis for declarations

The proof of soundness of cost analysis for function declarations follows the structure of the corresponding one of the size analysis. We start by restating inclusiveness for the type semantics with both sizes and costs.

Lemma 6.14 (Inclusiveness of $\mathcal{T}^{\mathsf{C}}_{\Sigma}$) *Let* $\{F_i : i \in \mathbb{N}\}$ *be an ascending chain, i.e.* $F_i \in [\mathbf{V} \to (\mathbf{V} \times \mathbb{N} \times \mathbb{N})_{\perp}]$ *and* $F_i \sqsubseteq F_{i+1}$ *for all* $i \in \mathbb{N}$*; if* $F_i \in \mathcal{T}^{\mathsf{C}}_{\Sigma}[\![\langle \forall \vec{\alpha}.\, \vec{\tau} \xrightarrow{s;h} \tau',\, \phi \rangle]\!]$ *then* $\bigsqcup_{i \in \mathbb{N}} F_i \in \mathcal{T}^{\mathsf{C}}_{\Sigma}[\![\langle \forall \vec{\alpha}.\, \vec{\tau} \xrightarrow{s;h} \tau',\, \phi \rangle]\!]$.

Proof sketch: The proof is analogous to the one for the size semantics (Lemma 5.16); the crucial property is that $(\mathbf{V} \times \mathbb{N} \times \mathbb{N})_{\perp}$ is a flat domain and therefore satisfies the ascending chain condition. □

Finally, we can state soundness of the cost analysis for function abstractions. Again the proof follows the structure of the one for size analysis (Theorem 5.23).

Theorem 6.15 (Soundness of cost analysis for declarations) *If* Σ *is consistent,* $\varphi \models \Gamma$ *and* $\Gamma, \Sigma \vdash_{\mathit{COST}} \mathit{decl} : \eta$, *then* $\mathcal{D}[\![\, \mathit{decl} \,]\!]\, \varphi \in \mathcal{T}^{\mathsf{C}}_{\Sigma}[\![\eta]\!]$.

Proof sketch: The proof is by case analysis on *decl* and is analogous to the one for the size analysis (Theorem 5.23). The soundness of the recursive functions relies on the inclusiveness of $\mathcal{T}^{\mathsf{C}}_{\Sigma}$ (Lemma 6.14). □

6.6 Optimisations

Our type and effect system can easily be extended to account for compiler optimisations provided these can be exposed in a syntax-directed way. We consider here some optimisations that reduce stack and heap space usage (and, indirectly, also time costs).

Optimisations are important for two reasons: first, they reduce the absolute time or space costs of Core Hume programs, thus making the language better suited for systems with limited resources. Secondly, space reuse optimisations can lower the cost complexity, making more algorithms exhibit linear or even constant space costs (Hofmann 2000). Thus, optimisations can have a positive impact on predictability of costs as well as on the performance.

6.6.1 Tail call optimisation

The standard *tail call optimisation* compiles a function call in a tail position into a direct jump, re-using the caller stack frame (Steele 1977, Jones and Johnsson 1987). In particular, a tail recursive function is optimised to run in constant stack space (i.e. an iterative loop).

Annotating tail calls

In order to distinguish tail calls for ordinary function calls we add an extra node to our abstract syntax for (annotated) expressions:

$$\widehat{e} \ ::= \ \cdots \quad | \quad f^{\downarrow k}\,(\widehat{e}_1, \ldots, \widehat{e}_n) \qquad \text{tail call } (k \geq 0)$$

A tail call is annotated with a parameter k specifying the number of stack-bound identifiers at the call point; these are no longer accessible after evaluation of the function arguments, and so can be deallocated before transferring control to f.

Example 6.16 The auxiliary list reverse function using an accumulating parameter can be written in tail-recursive fashion:

$$\begin{aligned}
\mathsf{letrec}\ \mathit{revAcc}\ (xs, ys)\ &= \mathsf{case}\ xs\ \mathsf{of} \\
&\quad\ \mathsf{Nil} \to ys \\
&\quad | \ \mathsf{Cons}\ (x, xs') \to \mathit{revAcc}^{\downarrow 4}\ (xs', \mathsf{Cons}\ (x, ys))
\end{aligned}$$

The alternative for Cons is a tail call to *revAcc*; the 4 stack entries associated with xs, ys, x and xs' can be removed *after* the evaluation of the arguments but *before* the recursive call.

Extending the operational semantics

To extend the Core Hume abstract machine with tail calls we introduce a new control directive $\mathsf{slide}(n,k)$ that shifts the top n stack values k positions down, reducing the stack depth by k; the transition rule for slide is:

$$\begin{aligned}&\langle \mathsf{slide}(n,k) : C,\, E,\, u_1 : \ldots : u_n : v_1 : \ldots : v_k : S,\, H\rangle \overset{r}{\hookrightarrow} \\ &\qquad\qquad\qquad \langle C,\, E,\, u_1 : \ldots : u_n : S,\, H\rangle\end{aligned} \tag{6.72}$$

The transition rule for evaluating a tail call $f^{\downarrow k}\ (\widehat{e}_1,\, \ldots,\, \widehat{e}_n)$ is similar to an ordinary application except that: 1) we do *not* push a continuation; and 2) we slide the function arguments before evaluating the function body.

$$\frac{(f\ (x_1, \ldots, x_n) = e') \in P}{\begin{aligned}\langle \mathsf{eval}(f^{\downarrow k}\ (e_1, \ldots, e_n) : C),\, E,\, S,\, H\rangle \overset{r}{\hookrightarrow} \langle &\mathsf{eval}(e_n) : \ldots : \mathsf{eval}(e_1) : \\ \mathsf{slide}(n,k) : \mathsf{fbind}(x_1 \ldots x_n) : &\mathsf{eval}(e') : \mathsf{ret}(n) : [],\, E,\, S,\, H\rangle\end{aligned}} \tag{6.73}$$

Extending the type and effect analysis

To extend the cost analysis for tail calls, we first extend the instrumentation function $\mathcal{C}_E$ that assigns cost annotations to expressions. The definition of $\mathcal{C}_E$ for ordinary function calls is (reproduced from Table 6.7):

$$\mathcal{C}_E[\![f\ (\widehat{e}_1, \ldots, \widehat{e}_n)]\!] \overset{\text{def}}{=} \mathsf{stack}^{1+n} f\ (\mathsf{stack}^{-1}\, \mathcal{C}_E[\![\widehat{e}_1]\!], \ldots, \mathsf{stack}^{-n}\, \mathcal{C}_E[\![\widehat{e}_n]\!])$$

The modification required for tail calls are:

$$\mathcal{C}_E[\![f^{\downarrow k}\ (\widehat{e}_1, \ldots, \widehat{e}_n)]\!] \overset{\text{def}}{=} \mathsf{stack}^{n} f^{\downarrow k}\ (\mathsf{stack}^{-1}\, \mathcal{C}_E[\![\widehat{e}_1]\!], \ldots, \mathsf{stack}^{-n}\, \mathcal{C}_E[\![\widehat{e}_n]\!]) \tag{6.74}$$

As would be expected, the stack cost for the tail call is one less than ordinary calls because no continuation is pushed. However, the number k of deallocated stack is *not* accounted in (6.74); this is because stack is deallocated *after* the evaluation of the arguments $\widehat{e}_1,\, \ldots,\, \widehat{e}_n$.

To account for stack deallocation, we introduce a specialised typing rule for tail calls. The typing rule for ordinary function calls is (reproduced from Table 6.8):

$$\frac{\begin{gathered}\Gamma \vdash_{\mathsf{INST}} f : \langle (\tau_1, \ldots, \tau_n) \xrightarrow{s_1 ; h_1} \tau,\, \phi_1\rangle \\ \Gamma \vdash_{\mathsf{COST}} (\widehat{e}_1, \ldots\ \widehat{e}_n) : (\tau_1, \ldots, \tau_n)\ \$\ s_2\, ; h_2 \mid \phi_2\end{gathered}}{\begin{aligned}\Gamma \vdash_{\mathsf{COST}} f\ (\widehat{e}_1, \ldots, \widehat{e}_n) : \tau\ \$\ s\, ; h \mid \phi_1 \wedge \phi_2 \wedge s = \max(s_1,\, s_2) \\ \wedge\, h = h_1 + h_2\end{aligned}}$$

The new rule for tail calls is:

$$\frac{\Gamma \vdash_{\mathsf{INST}} f : \langle(\tau_1, \ldots, \tau_n) \xrightarrow{s_1 ; h_1} \tau,\, \phi_1\rangle \qquad \Gamma \vdash_{\mathsf{COST}} (\widehat{e}_1, \ldots, \widehat{e}_n) : (\tau_1, \ldots, \tau_n) \mathbin{\$} s_2 \,; h_2 \mid \phi_2}{\Gamma \vdash_{\mathsf{COST}} f^{\downarrow k}\, (\widehat{e}_1, \ldots, \widehat{e}_n) : \tau \mathbin{\$} s \,; h \mid \phi_1 \wedge \phi_2 \wedge s = \max(s_1 - k,\, s_2) \wedge h = h_1 + h_2} \tag{6.75}$$

Rule (6.75) subtracts k from the latent stack cost s_1 but not from the stack cost s_2 of the arguments; this mimics the operational semantics of deallocating the stack after evaluating the arguments but before the function body.

6.6.2 Unboxed data types

The abstract machine of Section 6.2 represents values of an algebraic data type

$$\mathsf{data}\ \mathsf{D}\ \vec{\alpha} \;=\; c_1\ \vec{\tau}_1 \ \mid\ c_2\ \vec{\tau}_2 \ \mid\ \cdots \ \mid\ c_n\ \vec{\tau}_n$$

as a heap cell with a tag c_i and $|\vec{\tau}_i|$ arguments (where $|\vec{\tau}_i|$ is the arity of the constructor c_i).

For a data type where all constructors have zero arity (i.e. a pure sum type) such representation is wasteful: a value can be represented in a register or the stack by the constructor tag alone. Dually, a data type with a single constructor (i.e. a tuple) can be represented by its components in registers or the stack. Such “unboxing” of the data representation is an important optimisation technique in modern implementations of functional languages, e.g. the Glasgow Haskell Compiler (Jones 1992).

We will consider here the first of these optimisations, namely unboxing the representation of pure sum types; this avoids heap allocations in quite common situations (e.g. the boolean data type or any other enumerations).

Note that, for technical reasons, we will *not* deal with unboxing tuples. Although this could be implemented by allowing functions to return more than result on the stack, it would invalidate the invariant of Lemma 6.2 of our abstract machine and, consequently, the instrumented big-step evaluation semantics of Section 6.2.5 which is the formal basis for the stack bounds obtained by our cost analysis.

Representing unboxed data types

To modify the core language and abstract machine for this optimisation it suffices to introduce distinct syntax nodes for unboxed constructors and allow tags as unboxed values.[5] The formal changes are:

$$\begin{array}{lll} e & ::= \cdots & \text{expressions (Section 4.2.1)} \\ & \mid\ c\# & \text{unboxed constructor} \\ & \mid\ \mathsf{case\#}\ e_0\ \mathsf{of}\ \{c_i \to e_i\}_{i=1}^{n} & \text{unboxed case} \\ u & ::= \cdots & \text{unboxed values (Section 6.2)} \\ & \mid\ c & \text{constructor tag} \end{array}$$

Note that case# expressions do not bind variables because unboxed values cannot have arguments (i.e. are simple enumerations).

We require that the representation of a data type is uniform, i.e. that all constructors of the data type are represented in either boxed or unboxed form; this is to avoid the need for runtime tags on unboxed values to distinguish addresses from constructors. Therefore, the above syntax for expressions distinguishes an unboxed constructor $c\#$ from a boxed constructor with zero arguments $c\ ()$.

Extending the operational semantics

The small-step semantics of Section 6.2, Table 6.1 can easily be extended to handle unboxed data types. We introduce a new control directive select#($alts$) to perform selection on an unboxed constructor and three extra transitions:

$$\langle \mathsf{eval}(c\# : C),\, E,\, S,\, H\rangle \overset{r}{\hookrightarrow} \langle C,\, E,\, c : S,\, H\rangle \tag{6.76}$$

$$\langle \mathsf{eval}(\mathsf{case\#}\ e\ \mathsf{of}\ alts) : C,\, E,\, S,\, H\rangle \overset{r}{\hookrightarrow} \langle \mathsf{eval}(e) : \mathsf{select\#}(alts) : C,\, E,\, S,\, H\rangle \tag{6.77}$$

$$\frac{(c \to e) \in alts}{\langle \mathsf{select\#}(alts) : C,\, E,\, c : S,\, H\rangle \overset{r}{\hookrightarrow} \langle \mathsf{eval}(e) : C,\, E,\, S,\, H\rangle} \tag{6.78}$$

Evaluating an unboxed constructor (rule (6.76)) simply places the tag on top of the stack. Rule (6.77) evaluates the unboxed case by evaluating the discriminant first. Finally, rule (6.78) dispatches the correct alternative based on the constructor tag on the top of the stack; note that unlike rule (6.10) for boxed value selection, no static return is needed because unboxed case expressions do not bind variables.

Extending the type and effect analysis

To extend the type and effect analysis for unboxed constructors, it is enough to modify the cost annotations function $\mathcal{C}_E$: the construction of an unboxed value

[5] This would be straightforward in an real implementation because tags can be represented by machine integers.

incur a stack cost but no heap cost; and case analysis incurs no stack or heap costs.

$$\mathcal{C}_E[\![\, c\# \,]\!] \stackrel{\text{def}}{=} \mathsf{stack}^1\ c\ ()$$

$$\mathcal{C}_E[\![\, \mathsf{case}\#\ e_0\ \mathsf{of}\ \{c_i \to e_i\}_{i=1}^n \,]\!] \stackrel{\text{def}}{=} \mathsf{case}\ \mathcal{C}_E[\![e_0]\!]\ \mathsf{of}\ \{c_i\ () \to \mathcal{C}_E[\![e_i]\!]\ \}_{i=1}^n$$

The type and effect rules for annotated expressions of Table 6.8 remain unchanged.

6.6.3 Explicit heap deallocation

We now consider an optimisation for explicit heap deallocation. Following (Hofmann 2000, 2002) and (Hofmann and Jost 2006) we do so by introducing a case expression that deallocates the heap value that it matches against; such deallocation is safe if the value is not used in the continuation of the evaluation. The extended syntax is:

$$e\ ::=\ \cdots\ \mid\ \mathsf{case!}\ e_0\ \mathsf{of}\ \{c_i\ \vec{x}_i \to e_i\}_{i=1}^n \qquad \text{deallocating case expression}$$

Denotationally, case! has the same semantics as an ordinary case; operationally, case! performs deallocation of the result of e_0 *before* evaluating one alternative, so that the associated heap cell can be re-used.

In the remaining section we discuss the modifications to our abstract machine and type and effect analysis to implement this optimisation. We do not address the problem of checking if the use of case! is safe, i.e. determining the last use of a heap value.

Example 6.17 Assume that list constructors are heap allocated but booleans are not (e.g. by using the unboxing optimisation discussed in Section 6.6.2). Then the following program implements a destructive list insertion sort that re-uses the input list to construct the sorted one (Hofmann 2000):

$$\begin{aligned}
\mathsf{letrec}\ \mathit{insert}\ x\ \mathit{ys}\ &= \mathsf{case!}\ \mathit{ys}\ \mathsf{of}\\
&\quad [\,] \to x : [\,]\\
&\mid\ y : \mathit{ys}' \to \mathsf{if}\ x < y\ \mathsf{then}\ x : y : \mathit{ys}'\ \mathsf{else}\ x : \mathit{insert}\ x\ \mathit{ys}'\\
\mathsf{letrec}\ \mathit{isort}\ \mathit{xs}\ &= \mathsf{case!}\ \mathit{xs}\ \mathsf{of}\\
&\quad [\,] \to [\,]\\
&\mid\ x : \mathit{xs}' \to \mathit{insert}\ x\ (\mathit{isort}\ \mathit{xs}')
\end{aligned}$$

Either branch of *insert* deallocates a nil or a cons cell before evaluating the expressions on the right hand side. By induction on the length of the list, it is easy to verify that the net effect of *insert* is to grow the heap by a single cons cell (regardless of the list length). Because one cons cell is deallocated before each insertion, *isort* reuses the input list to construct the result—effectively performing the sort operation "in-place". □

More generally, Hofmann (2000) presented a linear type system to detect such safe re-uses of heap values and shows that linearly-typed first-order functional programs can be translated into `malloc`-free C code.

Operational semantics

To extend the operational semantics for deallocation we introduce a new control directive $\mathsf{select!}(\mathit{alts})$ and two new small-step reduction rules:

$$\langle \mathsf{eval}(\mathsf{case!}\ e\ \mathsf{of}\ \mathit{alts}) : C,\, E,\, S,\, H\rangle \overset{r}{\hookrightarrow} \langle \mathsf{eval}(e) : \mathsf{select!}(\mathit{alts}) : C,\, E,\, S,\, H\rangle \tag{6.79}$$

$$\frac{\begin{array}{cc} a\ \mathbf{at}\ r & H(a) = \langle c, u_1, \ldots, u_n\rangle \\ H' = H - a & (c\ (x_1, \ldots, x_n) \to e) \in \mathit{alts} \end{array}}{\begin{array}{l}\langle \mathsf{select!}(\mathit{alts}) : C,\, E,\, a : S,\, H\rangle \overset{r}{\hookrightarrow} \langle \mathsf{bind}(x_1,\, \ldots,\, x_n) : \mathsf{eval}(e) : \\ \qquad\qquad\qquad\qquad \mathsf{sret}(n, E) : C,\, E,\, u_1 : \ldots : u_n : S,\, H'\rangle\end{array}} \tag{6.80}$$

Rule (6.80) implements the selection and deallocation of a heap value whose address is on top of the stack; except for deallocation it is identical to the corresponding rule (6.10) for ordinary case selection.

In the above rule we write $H - a$ for the result of deallocating a single cell at address a from H. We assume that deallocation decreases heap residency by the space associated with the heap cell, i.e. $|H-a| = |H| - |\langle c, u_1, \ldots, u_n\rangle| = |H|-1-n$. Note also that the arguments $u_1, \ldots, u_n$ are *not* deallocated.

In order to be able to re-use the released heap space in subsequent evaluations, we allow only deallocation of value residing in the output region of evaluation; this is imposed by the side condition a **at** r in rule (6.80).

Extending the small-step semantics with deallocation means that the heap usage in an evaluation is no longer monotonically increasing (so Lemma 6.1 no longer holds). This implies that our definition of the big-step evaluation must be revised. Concretely, we must:

1. augment the transitive reduction relation to record both maximum stack and heap usage;
2. define both the stack and heap usage δ and γ in the big-step evaluation $H,\, E \vdash e \Downarrow_r u,\, H',\, \delta,\, \gamma$ relative to the maximum usage in intermediate configurations.

Let $\sigma \underset{M_s,\,M_h}{\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}} \sigma'$ be the transitive reduction from σ to σ' where M_s is the maximum stack and M_h the maximum heap. We define the big-step evaluation as:

$$\begin{aligned} &H,\, E \vdash e \Downarrow_r u,\, H',\, \delta,\, \gamma \overset{\text{def}}{\iff} \\ &\qquad \forall C\ \forall S\ \langle \mathsf{eval}(e) : C,\, E,\, S,\, H\rangle \underset{M_s,M_h}{\overset{r}{\hookrightarrow\!\!\!\!\rightarrow}} \langle C,\, E,\, u : S,\, H'\rangle \\ &\qquad\qquad \wedge\, \delta = M_s - |S| \wedge \gamma \geq M_h - |H| \end{aligned} \tag{6.81}$$

Note that we have relaxed the definition of heap usage in (6.81) compared to the previous one (Definition 6.3 in Section 6.2.5) by allowing an upper bound instead than just the exact cost. This is needed in order to prove the admissibility of following judgement:

$$\frac{\begin{array}{c} H_0,\, E \vdash e \Downarrow_r a,\, H_1,\, \delta_1,\, \gamma_1 \\ H_1(a) = \langle c, u_1, \ldots, u_n\rangle \qquad (c\ (x_1, \ldots, x_n) \rightarrow e') \in \mathit{alts} \\ H_1 - a,\, E[x_1 \mapsto u_1, \ldots, x_n \mapsto u_n] \vdash e' \Downarrow_r u',\, H_2,\, \delta_2,\, \gamma_2 \end{array}}{H_0,\, E \vdash \mathsf{case!}\ e\ \mathsf{of}\ \mathit{alts} \Downarrow_r u',\, H_2,\, \max(\delta_1, n + \delta_2),\, \max(\gamma_1,\, \gamma_1 + \gamma_2 - 1 - n)} \tag{6.82}$$

Note that we would need to prove (6.82) together with all the cases of Lemma 6.4 again for the revised definition (6.81); the proofs are analogous to the ones given before and we omit them.

Type and effect analysis

We now modify the type and effect analysis in two stages: first we modify the cost annotation function $\mathcal{C}_E$ to account for heap deallocation (6.83); and second, we introduce a extra typing rule for (annotated) case! expressions (6.84).

$$\begin{aligned} &\mathcal{C}_E[\![\, \mathsf{case!}\ e_0\ \mathsf{of}\ \{c_i\ \vec{x}_i \rightarrow e_i\}_{i=1}^n \,]\!] \overset{\text{def}}{=} \\ &\qquad \mathsf{case!}\ \mathcal{C}_E[\![\, e_0 \,]\!]\ \mathsf{of}\ \{c_i\ \vec{x}_i \rightarrow \mathsf{stack}^{|\vec{x}_i|}\, \mathsf{heap}^{-1-|\vec{x}_i|}\, \mathcal{C}_E[\![\, e_i \,]\!]\}_{i=1}^n \end{aligned} \tag{6.83}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\mathsf{COST}} \widehat{e}_0 : \tau'\ \$\ s_0\,;h_0 \mid \phi_0 \\ \Gamma \vdash_{\mathsf{INST}} c_i : \langle \vec{\tau}_i'' \rightarrow \tau',\, \phi_i'\rangle \qquad \vec{x}_i : \vec{\tau}_i'',\, \Gamma \vdash_{\mathsf{COST}} \widehat{e}_i : \tau\ \$\ s_i\,;h_i \mid \phi_i \qquad (\forall i) \end{array}}{\begin{array}{l} \Gamma \vdash_{\mathsf{COST}} \mathsf{case!}\ \widehat{e}_0\ \mathsf{of}\ \{c_i\ \vec{x}_i \rightarrow \widehat{e}_i\}_{i=1}^n : \tau\ \$\ s\,;h \mid \\ \qquad \phi_0 \wedge \bigvee_{i=1}^n (\phi_i \wedge \phi_i' \wedge s = \max(s_0,\, s_i) \wedge h = \max(h_0,\, h_0 + h_i)) \end{array}} \tag{6.84}$$

Note that, unlike case analysis for unboxed values of Section 6.6.2, we need a distinct typing rule for the deallocating case because the heap is no longer simply additive. The effect h_i associated with the heap cost of e_i can be negative (because of the negative annotations introduced by $\mathcal{C}_E$); however, at least h_0 available heap is always required to evaluate the case discriminant e_0; we therefore take the heap of the case expression as a whole to be the *maximum* of h_0 and $h_0 + h_i$.

We have implemented this extension in our cost analysis; it can infer zero heap cost for the in-place insertion sort (Example 6.17):

```
insert :: {Int^z0,[Int]^z1}-^z3^z4->[Int]^z2 | 1+z1=z2, 3>=z4,
  2+5*z1>=z3, z4>=0, z3>=1, z1>=0
isort :: [Int]^z0-^z2^z3->[Int]^z1 | z3=0, z0=z1, z2>=1,
  2+5*z0>=z2, 1+6*z0>=z2
```

Note that the inferred bound on the heap cost of *insert* is $z_4 \leq 3$ corresponding to the allocation of a single cons cell; and the heap cost of *isort* is $z_3 = 0$, i.e. *insort* re-uses the heap space of the input list.

Implementation issues

The introduction of heap deallocation means that we can no longer assume that the available memory is contiguous. One standard approach for managing the available heap is to represent it as a linked structure of free blocks (the *free-list*) (Knuth 1973). For the simple case when all blocks have the same size, allocation and deallocation can be implemented in a LIFO-fashion, with constant-time access costs and no fragmentation. Two problems arise with variable-sized blocks:

1. allocation and deallocation of a block require traversing the free-list, i.e. they are no longer constant-time operations;
2. the allocation of a contiguous block may fail even though smaller blocks totalling the required size are available, i.e. the runtime system may run out of heap due to *external fragmentation.*

The first of these problems leads to potential unpredictability of time costs which is undesirable for the intended application domain of real-time embedded systems; the second problem leads to a potential heap underflow that would be undetected by our static analysis.

Both these problems can be avoided by simply allocating blocks in a single size (i.e. the size of the largest heap cell); this trades heap space waste due to internal fragmentation for predictability. While this may seem overly wasteful, we remark that the heap cell sizes for Core Hume are determinable at compile-time from the types and that that we can consider the maximum size for each region separately.

Moreover, heap usage in Core Hume programs is likely to exhibit more regularity than that of a general-purpose language because region-resetting at the coordination layer guarantees that the heap is compacted at regular intervals. We therefore conjecture that it should be possible to obtain static bounds on fragmentation costs for allocation/deallocation of variable-sized blocks by combining some of the strategies in the literature, e.g. separate free lists by block size or the "buddy" systems (Wilson et al. 1995). We leave this subject open for further research.

6.7 Cost analysis for the coordination layer

We now discuss how to combine the cost analyses of separate functions into an analysis for a complete program, i.e. a network of Core Hume boxes. Our aim is to obtain static stack and heap cost bounds for each of the communication wires. Because these are the only dynamic memory requirements in Core Hume, this guarantees bounded space behaviour for the complete program.

Values in wires can be written and read independently of each other; we will therefore employ an independent attribute analysis that associates intervals of sizes and costs to each wire. This means that we need some mechanism to mediate between the linear constraints obtained by the expression analysis and the intervals in the coordination analysis; we introduce interval constraints with a *projection* operation for that purpose.

6.7.1 Interval constraints

An *interval* of integers is either empty $\perp$ or a pair $[l, r]$ of bounds $l \in \mathbb{Z} \cup \{-\infty\}$, $r \in \mathbb{Z} \cup \{+\infty\}$. The set **Interval** of integer intervals with the interval inclusion partial order $\sqsubseteq$ forms a complete lattice.

An *interval constraint* has the syntax $\ell \sqsupseteq \phi$, where ℓ is a variable and ϕ is a convex size formula (i.e. a conjunction of linear inequalities; see Section 5.5). Informally, a constraint $\ell \sqsupseteq \phi$ expresses a lower-bound on the range of ℓ as a projection of ϕ.

A *system* of interval constraints is a finite set $\{(\ell_i \sqsupseteq \phi_i)_{i=1}^{N}\}$ where the variables ℓ_i on the left-hand sides are not necessarily distinct. Systems of constraints always have a least solution with respect to the interval inclusion order (see Appendix A.2).

Restricting ϕ in an interval constraint $\ell \sqsupseteq \phi$ to be convex does not limit expressiveness: quantifier elimination of an arbitrary size formula yields a disjunctive equivalent $\phi_1 \vee \phi_2 \vee \ldots \vee \phi_n$ where each ϕ_i is convex; and a constraint $\ell \sqsupseteq (\phi_1 \vee \phi_2 \vee \ldots \vee \phi_n)$ is equivalent to the system $\{\ell \sqsupseteq \phi_1, \ell \sqsupseteq \phi_2, \ldots, \ell \sqsupseteq \phi_n\}$.

6.7.2 Box analysis

Consider a generic declaration of a box b with n inputs, m outputs and k rules:[6]

$$\begin{array}{l} \mathsf{box} \quad b \quad (x_1 : \tau_1,\, \ldots,\, x_n : \tau_n) \quad (y_1 : \tau'_1,\, \ldots,\, y_n : \tau'_m) \\ \mathsf{unfair/fair} \\ \qquad (x_1,\, \ldots,\, x_n) \to (e_{11},\, \ldots,\, e_{1m}) \\ \qquad\qquad \vdots \\ \qquad (x_1,\, \ldots,\, x_n) \to (e_{k1},\, \ldots,\, e_{km}) \end{array} \tag{6.85}$$

Assume also that input and output wires are associated with sized types with distinct annotations, i.e. that $\mathrm{FZV}(\tau_i) \cap \mathrm{FZV}(\tau'_j) = \emptyset$ and $i \neq j \implies \mathrm{FZV}(\tau_i) \cap \mathrm{FZV}(\tau_j) = \mathrm{FZV}(\tau_i) \cap \mathrm{FZV}(\tau_j) = \emptyset$ for all i, j.

We can perform size and cost analysis of each output separately because each output is associated with a separate heap region; and because the analysis will obtain upper bounds, each rule can be analysed separately.

Consider then the j-th output in the i-th rule of (6.85); using the type and effect analysis of Section 6.4 we obtain a cost analysis for e_{ij}:

$$x_1 : \tau_1,\, \ldots,\, x_n : \tau_n,\, \Gamma_0 \vdash_{\mathsf{COST}} e_{ij} : \tau'_j \ \$\ s_j\, ; h_j \mid \phi_{ij} \tag{6.86}$$

Γ_0 are the type assumption for the user-defined data constructor and functions and s_j, h_j are the stack and heap costs associated with the j-th output; the formula ϕ_{ij} expresses the output sizes and costs as a function of the input sizes. The coordination layer analysis for output j in rule i is then a set of lower bound constraints:

$$\begin{array}{r} \mathsf{LB}(i,j) \overset{\text{def}}{=} \{\, \ell \sqsupseteq \mathsf{SIMPLIFY}(\exists Y \setminus (X \cup \{\ell\}).\, \phi_{ij}) \ :\ \ell \in \mathrm{FZV}(\tau'_j) \cup \{s_j, h_j\} \\ \text{where } X = \bigcup_i \mathrm{FZV}(\tau_i) \text{ and } Y = \mathrm{FZV}(\phi_{ij}) \,\} \end{array} \tag{6.87}$$

Note that we employ the procedure SIMPLIFY of Section 5.5 to obtain a convex formula equivalent to $\exists Y \setminus (X \cup \{\ell\}).\, \phi_{ij}$. Finally, the analysis of box b is simply the union of the lower bounds obtained for all rules and outputs:

$$\mathsf{LB}(b) \overset{\text{def}}{=} \bigcup_{\substack{1 \le i \le k \\ 1 \le j \le m}} \mathsf{LB}(i,j) \tag{6.88}$$

Example 6.18 Consider the list copy box (Example 6.6, p. 173):

```
box copy (xs::[Int]) (ys::[Int])
```

[6] For simplicity we consider the case where the rule patterns are variables and the outputs are not *; the generalisations for these cases are straightforward.

```
match
 (xs) -> (copyList xs)
;

copyList :: [Int] -> [Int];
copyList [] = [];
copyList (x:xs) = x : copy_List xs;
```

The expression layer analysis for *copyList* yields

$$\Gamma_0 \overset{\text{def}}{=} \mathit{copyList} : \forall z_0 z_1 z_2 z_3.\, \langle \mathsf{List}^{z_0}\,\mathsf{Int} \xrightarrow{z_2;z_3} \mathsf{List}^{z_1}\,\mathsf{Int}, \\ z_0 = z_1 \wedge 1 + 3z_0 = z_3 \wedge 1 + 4z_0 \geq z_2 \wedge z_2 \geq 1\rangle \tag{6.89}$$

For the coordination analysis of the *copy* box, we assign distinct sizes to the input and output:

$\mathsf{List}^{i}\,\mathsf{Int}$	input xs
$\mathsf{List}^{o}\,\mathsf{Int}$	output ys
s	stack cost of ys
h	heap cost of ys

The analysis of the single rule yields:

$$xs : \mathsf{List}^{i}\,\mathsf{Int}, \Gamma_0 \vdash_{\mathsf{COST}} \mathit{copyList}\ xs : \mathsf{List}^{o}\,\mathsf{Int}\ \$\ s\,;h \mid \\ i = o \wedge 1 + 3i = h \wedge 3 + 4i \geq s \wedge s \geq 3 \tag{6.90}$$

Note that the expression stack cost in (6.90) is two words higher than the latent cost in (6.89) to account for the for the application. The analysis of the *copy* box is then:

$$\mathsf{LB}(\mathit{copy}) = \{\, o \sqsupseteq (i = o \wedge i \geq 0),\ h \sqsupseteq (1 + 3i = h \wedge i \geq 0), \\ s \sqsupseteq (3 + 4i \geq s \wedge s \geq 3)\,\} \tag{6.91}$$

The interval constraints (6.91) express the output size, stack and heap costs as functions of the input list size i. □

We remark that the assumption $\mathrm{FZV}(\tau_i) \cap \mathrm{FZV}(\tau'_j) = \emptyset$ that size variables in input and output types each box are distinct is needed for the soundness of the type and effect analysis in (6.86) (Theorem 6.13). However, this prevents analysing a box that connects an output to an input (i.e. a direct feedback loop) since in that situation both ports share the same wire.

Consider a box with a input i connected to output j; we can extend the analysis for this situation by simply letting τ'_j be a size-renaming of the type τ_i for the input. We proceed as in the previous case and add interval constraints $\tau_i \sqsupseteq \tau'_j \cup \{s_i \sqsupseteq$

s_j, $h_i \sqsupseteq h_j\}$ between the sizes and costs of input and output wires; the extension of $\sqsupseteq$ to sized types is simply the system of pairwise constraints between corresponding annotations.

6.7.3 Solving interval constraints

A solution of a system $S = \{(\ell_i \sqsupseteq \phi_i)_{i=1}^{N}\}$ of interval constraints is an assignment $\mathcal{V}$: **ZVar** $\to$ **Interval** from variables to integral intervals. Systems of interval constraints always have a least solution with respect to the interval containment order $\sqsubseteq$; the proof is presented in Appendix A.2. Furthermore, solutions can be obtained in a fully automated way using standard abstract interpretation techniques (see Appendix A.2 for details). Therefore the coordination layer analysis can obtain concrete stack and heap bounds that may be used in a compiler as static bounds for space allocation.[7]

Example 6.19 Consider the interval constraints (6.91) obtained for the analysis of the *copy* box. We can obtain finite bounds for a specific sizes by solving an augmented system with constraints specifying ranges for the input. For example, solving $\mathsf{LB}(\mathit{copy}) \cup \{i \sqsupseteq (0 \leq i \leq 10)\}$ we obtain the following solution:

$$\mathcal{V}(i) = \mathcal{V}(o) = [0,\, 10] \quad \mathcal{V}(s) = [3,\, 43] \quad \mathcal{V}(h) = [1,\, 31]$$

Thus, we get guaranteed worst-case bounds of 43 words for stack and 31 words of heap for the output wire *ys* for copying lists of length up to 10. □

Summary

In Sections 6.2 and 6.3 we presented a cost model for Core Hume program in the form of an abstract machine. This machine is the semantic basis used in defining notions of stack and heap costs for Core Hume programs.

In Section 6.4 we extended the sized type system of Chapter 5 with effects that approximate stack and heap costs of the Core Hume machine. We have also extended our core language with cost annotations that decouple the cost analysis from the specific cost model (Section 6.4.4) and introduced cost lifting transformations that allow trading analysis precision for lower time (Section 6.4.6).

[7] Subject, of course, to the expression layer analysis being able to infer upper bounds; otherwise, the coordination analysis obtains a stack or heap bound $+\infty$ which is safe but uninformative.

In Section 6.5 we proved the correctness of the cost analysis with respect to a cost-instrumented semantics.

In Section 6.6 we have shown how to extend the cost analysis to deal with some standard optimisations: tail calls, unboxed enumerations and explicit heap deallocation.

Finally, we have shown in Section 6.7 how to transpose the cost analysis of expressions to obtain concrete stack and heap bounds for the communication layer of Core Hume.

Chapter 7

Experimental results

In this chapter we present experimental results obtained using the size and cost analysis that were developed in Chapters 5 and 6. In order to assess the quality of the cost approximations, we will compare results obtained by our cost analysis with profiling information obtained from an implementation of the abstract machine described in Section 6.2.

The presentation is as follows: Section 7.1 discusses objectives and methodology. Section 7.2 presents the results of applying cost analysis to some standard functional algorithms and data structures. Section 7.3 presents results for some simplified embedded systems.

7.1 Objectives and methodology

It is well-known that establishing soundness of the approximations computed by a program analysis is not sufficient to judge its usefulness: an analysis that always produces a "don't know" answer will be trivially sound, but not useful. Therefore, in order to assess the usefulness of the cost analysis of Chapter 6, we will apply it to a small but representative set of Core Hume programs. The cost approximations can then be compared with actual profiling information obtained from an implementation of the abstract machine of Section 6.2.

However, since Hume is a research language, there are as yet no real-world applications written in Hume. Furthermore, our analysis and cost model are defined only for a subset of Hume that lacks many useful features for realistic applications (e.g. input-output, exceptions, more primitive types, etc). We have therefore chosen to evaluate the analysis using synthetic benchmarks divided in two groups:

1. some standard algorithms on functional data structures (e.g. lists and binary trees);
2. some prototype embedded systems modelled as Hume processes that react to external stimuli and must run *indefinitely* in bounded space.

The benchmarks in the first group are not complete Hume programs. Instead, they are intended to be representative of a repertoire of algorithms and data structures that a Hume programmer has at his or her disposal. By assessing the quality of the heap and stack bounds inferred by our analysis, we intend to demonstrate its usefulness for obtaining bounded space guarantees in a general functional programming context.

The second group of benchmarks consists of simplified reactive systems which employ recursive algorithms and data-structures. These will be used to demonstrate the applicability, in principle, of our cost analysis to bounding stack and heap space for complete Hume systems.

We will also consider some pragmatic aspects of the analysis, in particular, the running time and memory consumption. Since we are performing analysis at compile-time this has no impact on the deployed code, but must be efficient enough in order to be practical.

Finally, we point out that a web interface for our prototype implementation of the analysis available at `http://www.ncc.up.pt/~pbv/cgi/cost.cgi` allows experimenting with these and other examples.

7.2 Functional algorithms and data structures

In this section we will present the analysis for some standard algorithms on functional data structures such as lists and trees.

7.2.1 Lists

Our initial examples operate on list structures. We employ a Haskell-style syntax for the lists type with a default length measure, i.e. the (implicit) sized data type declaration is:

```
data [a]^n = []           { n=0 }
           | a : [a]^k    { n=1+k, 0<=k }
```

The constraint $0 \leq k$ in the cons alternative states that the list length of the tail is non-negative. Non-negativity constraints like these allow the size analysis to obtain more precise information (e.g. when matching a pattern like `x:xs` the analysis concludes that the list has at least one element).

The first example is the canonical list append function. Size type inference for this example was already presented in Chapter 5; we will therefore focus on the results of cost analysis. Our implementation allows pattern-matching equations that get translated into simple case expressions; the concrete syntax for append is:

```
append :: {[a],[a]} -> [a] ;
append [] ys = ys ;
append (x:xs) ys = x : append xs ys  ;
```

The output of cost analysis is a type signature annotated with variables and a constraint. We will present the straight output of the analysis in the same typewriter font as the concrete Hume program:

```
append :: {[a]^z1,[a]^z2}-^z4^z5->[a]^z3 | 3*z1=z5, z1+z2=z3,
  1+5*z1>=z4, z4>=1
```

Size and cost variables are named z_1, z_2, …etc. Size annotations on data types are preceded by a caret (`^`). Thus, z_1 is the size of the first list argument, z_2 is the size of the second list argument and z_3 is the size of the result. $\boldsymbol{\omega}$-annotations are omitted: for example, Int^{ω} is written as `Int`.

An arrow type carries annotations for the stack and heap costs of the corresponding function; the association is positional: z_4 corresponds to the stack cost and z_5 to the heap cost.

The size and cost constraint is simplified using the algorithms of Section 5.5. The constraint is in quantifier-free disjunctive form, with conjunctions written as commas (`,`) and disjunctions as double vertical bars (`||`); the former bind more tightly than the later, so no parenthesis are needed. All variables occurring in the constraint are free and must occur in the annotated type.

The constraints inferred for recursive functions are always convex, i.e. a conjunction of linear (in)equalities (as in the *append* example above). This is a consequence of using hulling and widening during the fixed point approximation (algorithm FIX of Section 5.5). For non-recursive functions we have the option of not applying the convex hull and instead obtaining a more precise disjunctive constraint. Our inference algorithm defaults to applying the convex hull even in these cases, but allows the programmer to specify non-hulling for specific functions. This behaviour yields a simpler constraint as the default (which is often precise enough), and allows higher precision only when required (we will present an example of this later on).

We have seen in Section 5.5 that our inference algorithm obtains the exact relation $z_1 + z_2 = z_3$ between input and output sizes for *append*. For the heap cost, we also get an exact equation $3z_1 = z_5$ expressing the relation between heap allocations and the size of the first input list: append allocates one new Cons cell (3 heap units in our cost model) for each element in the first argument list. For stack cost, we get upper and lower bound inequations $1 + 5z_1 \geq z_4 \geq 1$ relating the maximum stack depth to the length of the first list (as expected, since append recurses on this argument).

7.2.2 List reversal

Naive reverse

The naive list reversal function constructs the reversed list by successively appending singleton lists.

```
nrev :: [a] -> [a] ;
nrev [] = [] ;
nrev (x:xs) = append (nrev xs) [x] ;
```

The analysis of *nrev* requires a type assumption for *append*. Here we used the result obtained by analysing the previous example, but it could equally be obtained from a library of predefined functions and analyses (this modularity is one of the advantages of type-based program analyses). Using the type assumption for *append* obtained in the previous example, the analysis of *nrev* yields:

```
nrev :: [a]^z1-^z3^z4->[a]^z2 | z1=z2, 2+z4>=7*z1, 1+6*z1>=z3,
  z3>=1, z4>=1+4*z1
```

The equation $z_1 = z_2$ captures the precise size relation: the reversed list must have the same length as the original list.

We obtain only lower bounds for the heap cost: $2 + z_4 \geq 7z_1$ and $z_4 \geq 1 + 4z_1$. This is to be expected: we have seen in the previous example that heap cost of append is (exactly) linear in the length of the first list. Since *nrev* calls *append* for each list element, its heap cost will be quadratic in the length. Because our analysis is based on a linear constraint solver, we do not obtain an upper bound. We do, however, get an upper bound $1 + 6z_1 \geq z_3$ of the stack depth as a linear function of the input list length.

Reverse with an accumulating parameter

It is well-known that we can transform naive reverse into a linear time algorithm by replacing calls to append with an accumulating parameter. Our analysis can obtain tight linear cost bounds for this version of reverse.

```
revAcc :: {[a],[a]} -> [a] ;
revAcc xs ys = case xs of
    [] -> ys
  | x:xs' -> revAcc xs' (x:ys) ;

reverse :: [a] -> [a] ;
reverse xs = revAcc xs [] ;
```

The analysis of reverse and the auxiliary function yields:

```
revAcc :: {[a]^z1,[a]^z2}-^z4^z5->[a]^z3 | 3*z1=z5, z1+z2=z3,
  1+5*z1>=z4, z4>=1
reverse :: [a]^z1-^z3^z4->[a]^z2 | z1=z2, 1+3*z2=z4, z2>=0,
  z3>=2, 4+5*z2>=z3
```

First, we remark that we still obtain the exact size relation $z_1 = z_2$ between input and reversed lists. We also get a relation $1 + 3z_2 = z_4$ expressing the heap cost of *reverse* as a linear function of the reversed list length (which, by the first equation, is equal to the input list length). This is the exact heap cost of replicating the original list structure (one Nil cell plus z_1 Cons cells).

Second, we get a stack bound $4+5z_2 \geq z_3$, again as a function of the input/output list length z_2. Third, we remark that the coefficient of z_2 in this bound is lower than the one obtained for naive reverse. Therefore, our analysis was able to prove an overall lower stack cost for the accumulating version of *reverse* compared to the naive version.

Reverse with tail call optimisation

We observe that the function *revAcc* is tail recursive, i.e. the result for the recursive case is a tail call. Tail calls can be optimised into jumps with no extra stack growth and, therefore, tail recursive functions be executed in constant stack space.

Following the extension described in Section 6.6.1, we explicitly annotate *recAcc* as a tail call and apply the analysis to predict costs of the optimised program. Recall that a tail call $f^{\downarrow k}\ (e_1, \ldots, e_n)$ is annotated with the number k of bound identifiers at the call point; the concrete syntax for tail calls is `f^k e1 ... eN`, where `k` is a natural constant. The modified program is:

```
revAcc :: {[a],[a]} -> [a] ;
revAcc xs ys = case xs of
    [] -> ys
  | x:xs' -> revAcc^4 xs' (x:ys) ;

reverse :: [a] -> [a] ;
reverse xs = revAcc^1 xs [] ;
```

We have written *recAcc* with a case expression rather than pattern matching equations to make the bound identifiers explicit: the tail recursive call to *revAcc* is annotated with frame size 4 for identifiers xs, ys, x and xs'. *Reverse* is also defined by a (non recursive) tail call to *revAcc*; here the stack frame holds a single bound identifier. The analysis obtains:

```
revAcc :: {[a]^z1,[a]^z2}-^z4^z5->[a]^z3 | 3*z1=z5, z1+z2=z3,
  4>=z4, 1+3*z1>=z4, z4>=1
reverse :: [a]^z1-^z3^z4->[a]^z2 | z1=z2, 1+3*z2=z4, z3>=1, z2>=0,
  2+3*z2>=z3, 5>=z3
```

As promised the stack costs for the tail-optimised functions are bounded by constants ($z_4 \leq 4$ for *revAcc*, $z_3 \leq 5$ for *reverse*). Note that we also obtain linear stack bound $z_3 \leq 2+3z_2$ for *reverse*; the conjunction of inequations implies $z_3 \leq \min(2+3z_2, 5)$; in fact, $2+3z_2 < 5$ only for $z_2 = 0$ which corresponds to an empty result and original lists. Therefore, the stack cost predicted for *reverse* is 1 for the empty list and 5 for the non-empty ones; this matches the actual cost obtained from profiling.

Shuffling lists

The functions *append*, *nrev* and *revAcc* of the previous examples were all written in primitive-recursive form: there is a base case for `[]` and a recursive case for `x:xs` with the recursive call on the argument `xs`.

Since our analysis is based on semantic properties of sizes rather than the syntactical shape of recursion, we can infer sized types for more general recursive definitions. The following function "shuffles" a list by reversing the argument at each recursive call; this example is due to Hughes et al. (1996).

```
shuffle :: [a] -> [a] ;
shuffle [] = [] ;
shuffle (x:xs) = x : shuffle (reverse xs) ;
```

The argument of the recursive call is not *xs* but *reverse xs* and, therefore, *shuffle* is not primitive-recursive. But the analysis can infer that the argument size in the recursive call is decreasing and obtains the following invariant (where the *reverse* is tail recursive):

```
shuffle :: [a]^z1-^z3^z4->[a]^z2 | z1=z2, z3>=1,
  3+4*z1>=z3, 1+6*z1>=z3
```

The equation $z_1 = z_2$ captures the exact relation between input and output list. We also get two linear upper bounds on the stack cost, namely $z_3 \leq 3+4z_1 \wedge z_3 \leq 1+6z_1$; these are linear on the list size z_1 because the stack cost for each call to *reverse* is constant. However, no upper-bounds are obtained for heap costs; this is as would be expected because the worst-case costs are quadratic on the list size and, therefore, no linear upper-bound exists.

7.2.3 Take and drop

The next examples are the standard *take* and *drop* functions from the Haskell prelude. These functions are defined by recursion over a pair of natural number and a list and exemplify how our analysis deals with more complex termination conditions.

Using inductive naturals

First, we begin with an inductive definition of natural numbers in Peano-style; the size measure is the natural number itself.

$$\begin{array}{llr} \mathsf{data\ Nat}^n = \mathsf{Zero} & & \{\, n = 0 \,\} \\ & \mid \mathsf{Succ\ Nat}^k & \{\, k \geq 0 \wedge n = 1 + k \,\} \end{array}$$

Take and drop can then be defined by case analysis on the arguments:

```
take :: {Nat,[a]} -> [a] ;
take Zero     []     = [] ;
take Zero     (x:xs) = [] ;
take (Succ n) []     = [] ;
take (Succ n) (x:xs) = x : take n xs ;

drop :: {Nat,[a]} -> [a] ;
drop Zero     []     = [] ;
drop Zero     xs     = xs ;
drop (Succ n) []     = [] ;
drop (Succ n) (x:xs) = drop n xs ;
```

The size and cost analysis for these two definitions yields:

```
take :: {Nat^z1,[a]^z2}-^z4^z5->[a]^z3 | 1+3*z3=z5, z4>=1, z3>=0,
  3+6*z3>=z4, z2>=z3, 2+z2+5*z3>=z4, z1>=z3, 1+z1+2*z2+3*z3>=z4
drop :: {Nat^z1,[a]^z2}-^z4^z5->[a]^z3 | 1>=z5,
  3+z1+5*z2>=5*z3+z4+2*z5, z5>=0, z4>=1, z3+z5>=1, z2>=z3,
  3+6*z2>=6*z3+z4+z5, z1+z3>=z2
```

As might be expected, the equations obtained are more complex than the previous examples. We obtain linear relations not just between sizes and costs but also *between* costs themselves, e.g. the inequation $3 + z_1 + 5z_2 \geq 5z_3 + z_4 + 2z_5$ establishes a relation between the stack and heap costs for *drop*. This is because our constraints are based on convex polyhedra inequations and are thus *fully-relational*: the equations obtained can involve any subset of annotation variables; this allows more precise information than an independent attribute analysis, where there is no interplay between information obtained for each component (Nielson et al. 1999, pages 250–251). It is also possible to simplify the information obtained by performing existential quantification over the "irrelevant" variables followed by variable elimination. We will do so in the next paragraphs to interpret the results.

Size analysis Designate the constraints obtained above for *take* and *drop*, by ϕ_{take} and ϕ_{drop} respectively. We start eliminating variables z_4 and z_5 for stack and heap, thus obtaining size-only information.

$$\exists z_4. \exists z_5. \phi_{take} \simeq z_3 \geq 0 \wedge z_2 \geq z_3 \wedge z_1 \geq z_3 \tag{7.1}$$

$$\exists z_4. \exists z_5. \phi_{drop} \simeq z_3 \geq 0 \wedge z_2 \geq z_3 \wedge z_1 + z_3 \geq z_2 \tag{7.2}$$

Equation (7.1) states that the length of the result of *take* is smaller than the size of both the natural number and the argument list. Equation (7.2) states that the length of the result of *drop* is smaller than the length of the argument list, but by no more than the size of the natural number.

Heap analysis To focus solely on heap results, we eliminate variable z_4 associated with the latent stack cost.

$$\exists z_4.\, \phi_{take} \simeq 1 + 3z_3 = z_5 \wedge z_3 \geq 0 \wedge z_2 \geq z_3 \wedge z_1 \geq z_3 \tag{7.3}$$

$$\exists z_4.\, \phi_{drop} \simeq 1 \geq z_5 \wedge z_1 + z_3 \geq z_2 \wedge z_5 \geq 0 \wedge z_3 + z_5 \geq 1 \wedge z_2 \geq z_3 \tag{7.4}$$

From equation (7.3) we can verify that the heap cost of *take* is linear in the size of the output list z_3 (in fact, heap is only allocated for constructing the result list structure). Note that the equation $1+3z_3 = z_5$ gives the exact heap cost as a function of an unknown size z_3. However, z_3 is bounded by the inequations $z_2 \geq z_3 \wedge z_1 \geq z_3$. Thus, the heap usage is bounded as function of the size of the inputs.

Equation (7.4) yields a constant upper bound on heap consumption: $1 \geq z_5$. This is because, other than the Nil case, the result list is a part of the input list, thus no extra heap is allocated.

Stack analysis To focus on stack results, we eliminate variable z_5 associated with the latent heap cost.

$$\begin{gathered}\exists z_5.\, \phi_{take} \simeq z_4 \geq 1 \wedge z_3 \geq 0 \wedge 6 + 6z_3 \geq z_4 \wedge z_2 \geq z_3 \wedge \\ 2 + z_2 + 5z_3 \geq z_4 \wedge z_1 \geq z_3 \wedge 1 + z_1 + 2z_2 + 3z_3 \geq z_4\end{gathered} \tag{7.5}$$

$$\begin{gathered}\exists z_5.\, \phi_{drop} \simeq z_4 \geq 1 \wedge z_3 \geq 0 \wedge z_2 \geq z_3 \wedge 3 + 6z_2 \geq 6z_3 + z_4 \wedge \\ 2 + 6z_2 \geq 5z_3 + z_4 \wedge z_1 + z_3 \geq z_2 \wedge 1 + z_1 + 5z_2 \geq 3z_3 + z_4\end{gathered} \tag{7.6}$$

Equations (7.5) and (7.6) express upper-bounds relating the stack costs to linear combinations of z_3 (the result list size), z_1 and z_2 (the size of the arguments). But since z_3 is bounded by z_1 and z_2, this translates into stack upper-bounds as a linear functions of the size of the argument for both *take* and *drop*.

Using primitive integers

For the sake of efficiency we might prefer to use primitive (unboxed) integers rather than an inductive data type for naturals. Since case expressions are only applicable to boxed values (not integers), we rewrite the functions using a combination of boolean conditional, equality and arithmetic primitives:

```
take' :: {Int,[a]} -> [a];
take' n xs = case xs of
   [] -> []
|  x:xs' -> if n<=0 then [] else x:take' (n-1) xs' ;
```

```
drop' :: {Int,[a]} -> [a];
drop' n xs =  case xs of
   [] -> []
|  x:xs' -> if n<=0 then xs else drop' (n-1) xs' ;
```

Using suitable typing assumptions for the primitive operations, the cost analysis obtains:

```
take' :: {Int^z1,[a]^z2}-^z4^z5->[a]^z3 | 3*z5>=2+7*z3+z4,
  z5>=1+4*z3, 1+z2+3*z3>=z5, z4>=1, z3>=0, 2+4*z3>=z5
drop' :: {Int^z1,[a]^z2}-^z4^z5->[a]^z3 | 1+z2=z3+z5,
  4+5*z2>=5*z3+z4, z2>=z3, z4>=1, z3>=0, 1+5*z2>=2*z3+z4
```

The results for size analysis are:

$$\exists z_4 z_5.\, \phi_{take'} \simeq z_3 \geq 0 \wedge z_2 \geq z_3$$
$$\exists z_4 z_5.\, \phi_{drop'} \simeq z_3 \geq 0 \wedge z_2 \geq z_3$$

Comparing these results with (7.1) and (7.2) we can see that the size relations between z_1 and z_3 were lost. This is caused by the inability of the fixpoint iteration to infer the non-negativity of the integer argument.

We do obtain upper bounds for stack and heap costs expressed in terms of z_2 and z_3. Note that the heap costs are different; in particular, *drop'* does not execute in constant heap. This is caused by the allocation of a boolean value for the comparison at each recursive step. Of course, this extra heap could be avoided, e.g. by using the unboxing optimisation of Section 6.6.2.

7.2.4 List sorting

Our next examples are implementations of two classic sorting algorithms, namely *quicksort* and *mergesort*. For simplicity, we consider here sorting lists of integers using a primitive less-than-or-equal operator with zero heap and stack costs. However, the cost analysis should, in principle, be applicable other data types with a suitable comparison operation.[1]

The first versions of the sorting algorithms are purely functional, i.e. construct new lists for all intermediate results; this implies that the number of allocations will be supra-linear with respect to the list size and therefore our analysis will not obtain an upper-bound for the heap costs (it will, however, obtain bounds for sizes

[1] The more general solution (abstracting the comparison as a higher-order argument) is not possible since our language is first-order.

```
qsort :: [Int] -> [Int] ;
qsort [] = [];
qsort (x:xs)
   = case (split_by x xs) of
          (lo,hi) -> append (qsort lo) (x:qsort hi);

split_by :: {Int,[Int]} -> ([Int],[Int]) ;
split_by x [] = ([],[]);
split_by x (y:ys)
   = case (split_by x ys) of
          (lo,hi) -> if y<=x then (y:lo,hi) else (lo,y:hi) ;

append :: {[a],[a]} -> [a];
append [] ys = ys;
append (x:xs) ys = x:append xs ys;
```

Figure 7.1: Purely-functional Quicksort

and stack costs). In Section 7.2.5 we will present alternative versions using explicit deallocation to re-use heap space of the input and intermediate lists; our analysis will be able to infer linear bounds on the worst-case heap cost for those.

Quicksort

Figure 7.1 presents a Core Hume implementation of quicksort using an auxiliary function *split_by* that splits a list into the two sublists of lower and higher elements with respect to a "pivot" value. The main function *qsort* calls *split_by*, recursively sorts the two sublists and concatenates the results.

Our type and effect analysis infers the following annotated types for *split_by* and *qsort*:[2]

```
split_by :: {Int^z0,[Int]^z1}-^z4^z5->([Int]^z2,[Int]^z3) |
  5+7*z1=z5, z1=z2+z3, z4>=1, z1>=z2, z2>=0, 2+5*z1>=z4
qsort :: [Int]^z0-^z2^z3->[Int]^z1 | z0=z1, 12+5*z0+z3>=3*z2,
  1+8*z0>=z2, 12+z3>=19*z0, z2>=1, z3>=1+9*z0, 6+z3>=16*z0
```

We obtain information on the list lengths for both functions, but not for the integer sizes in lists; as discussed in Section 5.6.3, this is a limitation our type system

[2] We omit the analysis of *append* which was already presented in Section 7.2.1.

regarding collection types.

Equations $z_1 = z_2 + z_3 \wedge z_1 \geq z_2 \wedge z_2 \geq 0$ in the annotated type of *split_by* captures the exact size relation between the function's arguments and result: the sum of the lengths of the result lists equals the length of the argument list. The result lengths z_2 and z_3 are unknown but bounded by z_1 (because $z_1 = z_2 + z_3 \wedge z_1 \geq z_2 \wedge z_2 \geq 0$ entails $z_1 \geq z_2 \wedge z_1 \geq z_3$).

The stack and heap costs inferred for *split_by* are linear on z_1 (the size of the argument list): we get an exact linear equation $z_5 = 5 + 7z_1$ for the heap cost z_5, together with lower and upper bounds $1 \leq z_4 \leq 2 + 5z_1$ for the stack cost z_4.

The analysis of *qsort* obtains the equality relation $z_0 = z_1$ between the input and output lengths, i.e. *qsort* preserves the list size. Again, we get no size information on the list elements. The remaining equations express relations between sizes and costs. We get only lower bounds for the heap cost z_3; we would not expect an upper bound because the worst-case cost is quadratic in the list size z_0. We do get an upper bound $1 + 8z_0 \geq z_2$ for the stack depth z_2 as a function of the input list length z_0. In fact, we shall see that this matches the maximum stack depth obtain by profiling exactly (see Table 7.1).

Note that our analysis would infer less precise size and cost information for the standard textbook definition of quicksort using two separate filters with inverted conditions, e.g. as in Bird and Wadler (1988), Chakravarty and Keller (2002), Hutton (2007):

```
qsort [] = []
qsort (x:xs) = let lo = [x'<-xs | x'<=x]
                   hi = [x'<-xs | x'>x]
                in qsort lo ++ x:qsort hi
```

For such a definition our size analysis would infer $|lo| \leq |xs|$ and $|hi| \leq |xs|$ but it would *not* infer the relation $|lo| + |hi| = |xs|$. Consequently, only a lower bound would be obtained for the length of the sorted list $|qsort\ xs| \geq |xs|$ and no upper bound would be obtained for the stack cost.

Mergesort

Figure 7.2 presents the implementation of mergesort using auxiliary functions *split* to divide a list into two (approximate) halves and *merge* to combine two sorted lists in order. The main function *msort* splits the input list, recursively sorts each half and merges the results.

```
msort :: [Int] -> [Int] ;
msort [x] = [x] ;
msort xs = case (split xs) of
                (xs1,xs2) -> merge (msort xs1) (msort xs2) ;

split :: [a]  -> ([a],[a])  ;
split []  = ([], []);
split [x] = ([x], []);
split (x:y:t) = case (split t) of (xs,ys) -> (x:xs, y:ys) ;

merge :: {[Int],[Int]} -> [Int] ;
merge xs ys     -- assumes xs is not []
   = case xs of
      x:xs’ -> case ys of
               [] -> xs
             | y:ys’ -> if x<=y then x:merge ys xs’
                        else y:merge xs ys’ ;
```

Figure 7.2: Purely-functional Mergesort

Our implementation differs from the typical textbook presentation of mergesort, e.g. from Bird and Wadler (1988):

```
msort [x] = [x]
msort xs = merge (msort xs’) (msort xs’’)
   where n = length xs ‘div‘ 2
         xs’ = take n xs
         xs’’= drop n xs
```

Instead of using *length*, *take* and *drop*, we use a single *split* function that recursively breaks a list into two halves. This allows the analysis to infer a more precise size relation between the two halves; it also avoids the use of the division operation whose size relation cannot be expressed as an exact Presburger formula and therefore would admit only an uninformative sized type, e.g. $div : \langle (\mathsf{Int}^{\omega}, \mathsf{Int}^{\omega}) \rightarrow \mathsf{Int}^{\omega},\ \mathit{True} \rangle$.[3]

We also point out that *merge* assumes that its first argument is not the empty list. This invariant holds for the call to *merge* in *msort* and is maintained in the recursive calls by flipping the arguments, if necessary. Thus, it is sufficient to test only the

[3] Divisions by *constants* admit informative sized types, e.g. $div2 : \forall ij.\ \langle \mathsf{Int}^{i} \rightarrow \mathsf{Int}^{j},\ i-1 \leq 2j \leq i \rangle$. It would be possible to employ an initial partial evaluation phase to statically detect such uses.

second argument of *merge* for emptiness. This uniform termination condition allows our cost analysis to obtain more accurate stack bounds.

Our type and effect analysis infers the following annotated types for this implementation of mergesort:

```
split :: [a]^z1-^z4^z5->([a]^z2,[a]^z3) | z1=z2+z3,  z1>=z2,
  2*z2>=z1, 1+z1>=2*z2, 5+6*z1=3*z2+z5, 2+4*z1>=z2+z4,
  3+4*z1>=2*z2+z4,  z4>=1
merge :: {[Int]^z0,[Int]^z1}-^z3^z4->[Int]^z2 | z0+z1=z2, z0>=1,
  4*z0+4*z1>=4+z4, 12+7*z4>=4*z3, 4*z1+z4>=0, z4>=0, z3>=1
msort :: [Int]^z0-^z2^z3->[Int]^z1 | z0=z1, z0>=1,
  10*z0>=1+z2, 29+z3>=24*z0, 15+z3>=11*z0+z2, z2>=1, 7+z3>=7*z0+z2,
  20+z3>=21*z0, 14+z3>=18*z0
```

As with the *split_by* function in the quicksort example, the lengths z_2, z_3 of the output of *split* are not known exactly. But for *split* our analysis obtains much more precise ranges:

$$z_1 = z_2 + z_3 \wedge z_1 \geq z_2 \wedge 2z_2 \geq z_1 \wedge 1 + z_1 \geq 2z_2 \implies \begin{cases} z_1 = z_2 + z_3 \\ z_1/2 \leq z_2 \leq (1 + z_1)/2 \end{cases}$$

Thus, there are only two cases: if z_1 is even then $z_2 = z_3 = z_1/2$; if z_1 is odd, then $z_2 = (1 + z_1)/2$ and $z_3 = (-1 + z_1)/2$. The size analysis was able to infer more information because, unlike *split_by*, the behaviour of *split* depends on the lists lengths but not on the values in the list.

For the *msort* function we obtain the equality $z_0 = z_1$ between the lengths of input and output lists. As would be expected, we do not get an upper bound on the number of allocations for the purely functional *msort* (the worst-case bound is $O(n \log n)$ for a list of length n). We do, however, get a linear upper bound $1 + 10z_0 \geq z_2$ for on the stack depth in terms of the list length z_0.

7.2.5 Destructive list sorting

Both implementations of list sorting in the previous section allocate new lists for all intermediate results. This is rather wasteful of space when compared to a typical implementation in an imperative language, where the sorting could be performed in-place.

We can improve the space behaviour of the functional algorithms by explicitly deallocating the input and intermediate lists (using the optimisation Section 6.6.3). Moreover, because the optimisation makes the cost linear on the list size, this allows

```
qsort_d :: [Int] -> [Int] ;
qsort_d xs = case! xs of
     [] -> []
   | x:xs' -> case! (split_by_d x xs') of
              (l,r) -> append_d (qsort_d l) (x:qsort_d r) ;

split_by_d :: {Int,[Int]} -> ([Int],[Int]);
split_by_d x ys = case! ys of
     [] -> ([],[])
   | y:ys' -> case! (split_by_d x ys') of
              (lo,hi) -> if! y<x then (y:lo,hi)  else (lo,y:hi) ;

append_d :: {[a],[a]} -> [a];
append_d xs ys = case! xs of
                    [] -> ys
                  | x:xs' -> x : append_d xs' ys;
```

Figure 7.3: Destructive Quicksort

our analysis to obtain upper-bounds on the heap cost; Figures 7.3 and 7.4 present the modified *qsort_d* and *msort_d*, respectively. Note that, in order to avoid using deallocated values we sometimes reconstruct lists in recursive calls (e.g. in the *merge_d* function in Figure 7.4).

Note also that we do not consider here the optimisation for unboxing data values (Section 6.6.2) and so the boolean result of comparisons are allocated in the heap; these can be deallocated immediately after scrutiny. For readability, we write a case! expression over a boolean value as "if! . . . then . . . else . . . ".

Destructive Quicksort

The type and effect analysis of destructive quicksort yields the following annotated types:

```
append_d :: {[a]^z1,[a]^z2}-^z4^z5->[a]^z3 | z5=0, z1+z2=z3,
  1+5*z1>=z4, z4>=1
split_by_d :: {Int^z0,[Int]^z1}-^z4^z5->([Int]^z2,[Int]^z3) |
  z1=z2+z3, 4>=z5, 2+5*z1>=z4, z5>=0, z4>=1, z1>=z2, z2>=0
qsort_d :: [Int]^z0-^z2^z3->[Int]^z1 | z0=z1, z3>=0, z2>=1, z0>=z3,
  1+8*z0>=z2
```

For *append_d* the analysis infers the heap cost $z_5 = 0$, i.e. that the function does not allocate any new heap. In fact, the base case deallocates a `[]` but does no further allocation, so that the amount of available heap after execution of *append_d* is higher than initially available. However, to account for the worst-case in intermediate configurations, the heap cost is defined as the *maximum* difference between initial and intermediate states (not just the initial and final states), so it is always non-negative.

For the heap cost of *split_by_d* we get a constant upper bound $z_5 \leq 4$ (i.e. independent of the input list length). This cost is incurred in the alternative for the empty list by the deallocation of a single `[]` followed by the allocation of `([],[])`. The alternative for the non-empty list re-uses a cons and pair cell, so the net difference is zero.

For the *qsort* function itself the analysis obtains a linear upper bound $z_3 \leq z_0$ for the heap cost z_3. This is because each recursive step of *qsort_d* calls *split_by_d* but deallocates only a pair cell. Note that the *append_d* deallocates one `[]` for each recursive call, so that the final size of the heap will be the same as the initial; our analysis infers only the initially required heap but not the available heap after execution.

Destructive Mergesort

The type and effect analysis of destructive mergesort yields:

```
split_d :: [a]^z1-^z4^z5->([a]^z2,[a]^z3) | z1=z2+z3, 2*z2>=z1,
  4>=z5, 2+4*z1>=z2+z4, z5>=0, z4>=1, 1+z1>=2*z2, z1>=z2,
  3+4*z1>=2*z2+z4
merge_d :: {[Int]^z0,[Int]^z1}-^z3^z4->[Int]^z2 | z4=0, z0+z1=z2,
  z3>=1, z1>=0, z0>=1, 7*z0+7*z1>=3+z3
msort_d :: [Int]^z0-^z2^z3->[Int]^z1 | z0=z1, 10*z0>=z2, z2>=1,
  4+12*z0>=z2+2*z3, 4*z0>=4+z3, 5*z0>=2+2*z3, 16*z0>=12+z2, z3>=0
```

As in the previous example, the analysis infers a constant bound $z_5 \leq 4$ for the heap cost of *split_d*. This again is caused by the allocation of the pair in the alternative for `[]`.

For *merge_d* the analysis infers a zero heap cost $z_4 = 0$. As in *append_d*, *merge_d* will deallocate one more constructor than it allocates, leaving a larger amount of available heap after execution; this is not captured in the heap effect.

Finally, for *msort_d* we get two inequations $z_3 \leq 4z_0 - 4 \wedge 2z_3 \leq 5z_0 - 2$ giving upper-bounds on the heap cost; this implies that the heap cost is bounded by the

```
msort_d :: [Int] -> [Int] ;
msort_d xs = case! xs of
 x:xs' -> case! xs' of
          [] -> [x]
        | y:xs'' -> case! (split_d (x:y:xs'')) of
                    (xs1,xs2) ->
                      merge_d (msort_d xs1) (msort_d xs2) ;

-- merge two lists of integers in order
-- assumes the first list is not []
merge_d :: {[Int],[Int]} -> [Int] ;
merge_d xs ys
   = case! xs of
      x:xs' -> case! ys of
                [] -> x:xs'
              | y:ys' -> if! x<=y then x:merge_d (y:ys') xs'
                         else y:merge_d (x:xs') ys' ;

-- split a list into two halves
split_d :: [a]  -> ([a],[a])  ;
split_d xs = case! xs of
    [] -> ([],[])
  | x:xs' -> case! xs' of
             [] -> ([x],[])
            | y:t -> case! (split_d t) of
                     (t', t'') -> (x:t', y:t'') ;
```

Figure 7.4: Destructive Mergesort

minimum, i.e. $z_4 \leq \min(4z_0 - 4,\ \frac{5}{2}z_0 - 1)$ (where z_0 is the length of the input list. Note that, as in the *qsort_d* example, *msort_d* will release all the heap it allocates, but the inferred heap effect cannot capture this: it only approximates the required *initial* heap.

7.2.6 Binary search trees

Unlike the sized type systems of Hughes et al. and Chin and Khoo, we allow used-defined size measures; this is particularly relevant for non-linear data structures like trees, where choosing different sizes measures can be according to intended use can be beneficial.

We illustrate this issue with a simple example of binary search trees with labelled inner nodes and empty leaves. The underlying data type declaration is:

$$\mathsf{data\ Tree}\ a = \mathsf{Leaf}\ \mid\ \mathsf{Branch}\ (a,\ \mathsf{Tree}\ a,\ \mathsf{Tree}\ a)$$

We can augment this data type with a size measure counting the *number of inner nodes*:

$$\begin{array}{rll} \mathsf{data} & \mathsf{Tree}^n\ a & \\ = & \mathsf{Leaf} & \{\, n = 0\, \} \\ \mid & \mathsf{Branch}\ (a,\ \mathsf{Tree}^k\ a,\ \mathsf{Tree}^m\ a) & \{\, n = 1 + k + m\ \wedge \\ & & \quad 0 \leq k \wedge 0 \leq m\, \} \end{array} \tag{7.7}$$

Alternatively, we could choose the size measure to be the *maximum height* of the tree. Since the maximum is not included in our size constraint syntax, we need to write it using a disjunction of linear inequations:

$$\begin{array}{rll} \mathsf{data} & \mathsf{Tree}^h\ a & \\ = & \mathsf{Leaf} & \{\, h = 0\, \} \\ \mid & \mathsf{Branch}\ (a,\ \mathsf{Tree}^l\ a,\ \mathsf{Tree}^r\ a) & \{\, (0 \leq l \leq r \wedge h = 1 + r) \vee \\ & & \quad (0 \leq r \leq l \wedge h = 1 + l)\, \} \end{array} \tag{7.8}$$

Depending on the algorithm and cost metric, either measure could be more suitable. For example, for analysing a traversal algorithm that visits every node, we might choose size measure (7.7). Conversely, for a binary search algorithm we would chose size measure (7.8).

Instead of trying to choose the metric automatically, we leave it up to the programmer to specify it for new data types (or to leave the data type unsized). However, no unsound information comes from this choice: a less insightful choice will simply mean that the analysis will not yield as precise size or cost bounds as it might. In fact, it is possible to start with a simple but coarse measure and later refine it with the aim of improving the results of cost analysis.

Tree traversal

Our first example is a simple tree traversal algorithm that lists the labels in-order: for each node, first visit the left sub-tree, then the current label, and finally the right sub-tree. The visiting order is represented by the result list of labels.

We use an auxiliary accumulating parameter to avoid the need for concatenating lists. More importantly, the accumulating parameter makes the algorithm exhibit linear complexity on the number of nodes, which is necessary for our analysis to infer upper bounds.

```
inorder :: Tree a -> [a];
inorder t = inorderAcc t [];

inorderAcc :: {Tree a, [a]} -> [a];
inorderAcc Leaf xs = xs ;
inorderAcc (Branch x l r) xs = inorderAcc l (x:inorderAcc r xs);
```

Declaring the size of trees to be the number of inner nodes as in (7.7), the analysis yields the following:

```
inorderAcc :: {Tree^z1 a,[a]^z2}-^z4^z5->[a]^z3 | z1+z2=z3,
  3*z1=z5, 1+7*z1>=z4, z4>=1
inorder :: Tree^z1 a-^z3^z4->[a]^z2 | z1=z2, z2>=0, 1+3*z2=z4,
  z3>=2, 4+7*z2>=z3
```

The size analysis produces an exact result: the result list length z_2 must be equal to the number of nodes z_1 in the tree. Equation $1 + 3z_2 = z_4$ expresses the exact heap cost z_4; this corresponds to the allocation of the result list. The stack depth z_3 is bounded by a linear term on output size: $4 + 7z_2 \geq z_3$. Therefore, the analysis obtains upper bounds for both stack and heap costs.

We remark that the outermost recursive call to *inderAcc* is in a tail position and so we could be optimised as described in Section 6.6.1. This would lower the stack cost but, unlike the list reverse example, would *not* make *inorderAcc* run in constant stack because of the inner recursive call.

Finally, note that size polymorphism is needed for typing *inorderAcc* because this function is applied to arguments with different sizes in the body of the recursive definition. In fact, the result of the first call is the argument for the second call; without size polymorphism this would yield an unsatisfiable size constraint.

Insertion in order

The following example is the insertion of a value in a tree respecting a total ordering on labels. Analogously to what was done for the sorting algorithms, we conduct the analysis for the simple case of integer labels. To make the example more challenging, the label will not be inserted if it occurs already in the tree.

```
insert :: {Int,Tree Int} -> Tree Int ;
insert x t = case t of
       Leaf -> Branch x Leaf Leaf
     | Branch y l r ->
               if x<y then Branch y (insert x l) r
               else if y<x then Branch y l (insert x r)
               else t ;
```

It is clear that the number of recursive calls of *insert* is bounded by the tree height. We therefore choose the tree size measure of declaration (7.8). Our analysis obtains:

```
insert :: {Int^z0,Tree^z1 Int}-^z3^z4->Tree^z2 Int | 1+z1>=z2,
  z2>=z1, 14*z2>=8+4*z1+z4, 13+34*z1+6*z4>=34*z2+5*z3, z3>=1,
  6*z2>=z4, 20*z2>=8+2*z3+z4, 4*z1+z4>=2+4*z2, 2+6*z1+z4>=5*z2+z3
```

Designating by ϕ_{ins} the constraint above, we perform some simplifications to interpret the results. To obtain size information we eliminate variables z_3 and z_4 for stack and heap:

$$\exists z_3.\, \exists z_4.\, \phi_{ins} \simeq 1 + z_1 \geq z_2 \geq z_1 \wedge z_2 \geq 1$$

The inequations $1+z_1 \geq z_2 \geq z_1$ express the best possible interval for the result tree height z_2 as a function of the input tree height z_1: it is (at least) the input height and (at most) one plus the input height. The constraint $z_2 \geq 1$ expresses the fact that the result tree has a height of at least one, i.e. it can not be a leaf.

To obtain stack information we eliminate variable z_4, thus obtaining linear relations between stack cost and sizes (and *mutatis mutandis* for heap).

$$\exists z_4.\, \phi_{ins} \simeq \ldots \wedge 2z_1 + 5z_2 \geq z_3 \geq 1$$
$$\exists z_3.\, \phi_{ins} \simeq \ldots \wedge 6z_2 \geq z_4 \wedge 14z_2 - 4z_1 - 8 \geq z_4 \geq 2 - 4z_1 + 4z_2$$

7.2.7 Red-black balanced trees

The next example is an insertion algorithm into self-balancing trees (Guibas and Sedgewick 1978, Okasaki 1998). The data type is identical to binary trees except

```
-- red-black trees; size measure is the height
data Tree^h = Leaf { h=0 }
            | Branch Colour a (Tree^l a) (Tree^r a)
                { 0<=l, l<=r, h=1+r || 0<=r, r<=l, h=1+l } ;
data Colour = Red | Black ;

-- balanced insertion
insert :: {Int, Tree Int} -> Tree Int;
insert x t = case (insertRec x t) of
             Branch c y l r -> Branch Black y l r;
-- auxiliary recursive insertion
insertRec :: {Int, Tree Int} -> Tree Int;
insertRec x t
      = case t of
        Leaf -> Branch Red x Leaf Leaf
      | Branch c y l r ->
          if x<y then lbalance c y (insertRec x l) r
          else if y<x then rbalance c y l (insertRec x r)
          else t ;

-- left and right balancing function
lbalance :: {Colour, a, Tree a, Tree a} -> Tree a ;
lbalance Black z (Branch Red y (Branch Red x a b) c) d
        = Branch Red y (Branch Black x a b) (Branch Black z c d) ;
lbalance Black z (Branch Red x a (Branch Red y b c)) d
        = Branch Red y (Branch Black x a b) (Branch Black z c d) ;
lbalance c x l r = Branch c x l r;

rbalance :: {Colour, a, Tree a, Tree a} -> Tree a ;
rbalance Black x a (Branch Red y b (Branch Red z c d))
        = Branch Red y (Branch Black x a b) (Branch Black z c d) ;
rbalance Black x a (Branch Red z (Branch Red y b c) d)
        = Branch Red y (Branch Black x a b) (Branch Black z c d) ;
rbalance c x l r = Branch c x l r;
```

Figure 7.5: Insertion into a red-black balanced tree.

for the addition of a “red” or “black” tag at each inner node. Since we are going to analyse insertion, we chose the tree height as the size measure; the data type of colours is unsized. The sized type declaration is as follows:

$$\begin{array}{lll}
\mathsf{data} & \mathsf{Colour} = \mathsf{Red} \mid \mathsf{Black} & \\
\mathsf{data} & \mathsf{Tree}^{h}\ a = \mathsf{Leaf} & \{\, h = 0 \,\} \\
\mid & \mathsf{Branch}\ (\mathsf{Colour},\, a,\, \mathsf{Tree}^{l}\ a,\, \mathsf{Tree}^{r}\ a) & \{\, (0 \leq l \leq r \wedge h = 1 + r) \vee \\
 & & \quad (0 \leq r \leq l \wedge h = 1 + l) \,\}
\end{array}$$

Red-black trees respect two conditions: 1) no red node has a red child; 2) every path from the root node to a leaf has the same number of black nodes. Together these conditions imply that the maximum height a red-black tree with n nodes is $2\lfloor\log(n + 1)\rfloor$, i.e. the tree is balanced. The insertion algorithm performs some rebalancing transformations that ensure the two conditions are invariants.

Figure 7.5 presents the insertion algorithm of (Okasaki 1998) adapted to Core Hume. It makes use of some auxiliary functions: *insertRec* performs insertion recursively, while the *lbalance* and *rbalance* functions perform the re-balancing transformations on the left and right sub-trees, respectively.[4]

The balancing functions use nested patterns that must be translated into simple case expressions. Unlike the previous examples, the translation of these patterns leads to a significant expansion of code size. This fact, together with the disjunctions introduced by the size measure, leads to much larger constraints than previous examples, with a consequent negative impact on the analysis time. We therefore choose to employ the cost approximation transformations discussed in Section 6.4.4 to trade less precise cost bounds for a faster analysis and shorter constraints. The following results were obtained with a cost annotation depth zero (the least accurate approximation). We present results for sizes first and stack and heap later.

Size analysis An initial attempt using size analysis of the functions in Figure 7.5 yields:

```
lbalance :: {Colour,a,Tree^z1 a,Tree^z2 a}-^z4^z5->Tree^z3 a |
  z3>=z1, 2*z3>=2+z1+z2, 1+z1+z2>=z3, z3>=1+z2, z2>=0
rbalance :: {Colour,a,Tree^z1 a,Tree^z2 a}-^z4^z5->Tree^z3 a |
  2*z3>=2+z1+z2, z3>=1+z1, z1>=0, 1+z1+z2>=z3, z3>=z2
insertRec :: {Int^z0,Tree^z1 Int}-^z3^z4->Tree^z2 Int |
  z2>=1, z1>=0
```

[4] Our implementation includes the optimisation described in Exercise 3.10 of (Okasaki 1998) that separates the balancing functions for left and right sub-trees.

```
insert :: {Int^z0,Tree^z1 Int}-^z3^z4->Tree^z2 Int |
  z1>=0, z2>=1
```

The results are sound but uninformative: the size constraint $z_2 \geq 1$ for the result of *insert* gives only a lower bound on the size (at least one, meaning that the result tree is not empty).

The cause of this coarse approximation is the analysis of *lbalance* and *rbalance*: in either case, the inequality $1+z_1+z_2 \geq z_3$ is overestimating the height of the result tree to be (one plus) the *sum* of the heights z_1, z_2. But the balancing functions will never concatenate the two trees along the longest path (in fact, this would result in an *unbalanced* tree).

We can get a better bounds if we instruct the analysis to avoid computing the convex hull of the size formula; this is done by an analysis directive `pragma "nohull"` before the definitions *lbalance* and *rbalance*. Running the analysis again yields:

```
lbalance :: {Colour,a,Tree^z1 a,Tree^z2 a}-^z4^z5->Tree^z3 a |
  1+z2=z3, z1>=0, z2>=z1 || 1+z1=z3, z2>=0, z3>=1+z2 ||
  2+z2=z3, z1>=2, z3>=z1 || 1+z1>=z3, z3>=z1, z2>=0, z1>=2,
  z3>=2+z2
rbalance :: {Colour,a,Tree^z1 a,Tree^z2 a}-^z4^z5->Tree^z3 a |
  1+z2=z3, z1>=0, z2>=z1 || 1+z1=z3, z2>=0, z3>=1+z2 ||
  1+z2>=z3, z3>=z2, z1>=0, z2>=2, z3>=2+z1 || 2+z1=z3, z2>=2,
  z3>=z2
insertRec :: {Int^z0,Tree^z1 Int}-^z3^z4->Tree^z2 Int |
  z2>=1, 1+z1>=z2
insert :: {Int^z0,Tree^z1 Int}-^z3^z4->Tree^z2 Int |
  z2>=1, 1+z1>=z2
```

The constraints obtained for *lbalance* and *rbalance* are now a disjunction of conjunctions that describe the size changes more precisely by distinguishing relations among the heights of the sub-trees. As a result, we get a more precise size relation for the recursive insertion *insertRec* (and consequently, for the top level function *insert*): $1 + z_1 \geq z_2$ is the expected upper-bound on the height of the resulting tree (it is at most one plus the original tree height). Keeping the disjunctive form in the constraints inferred for *lbalance* and *rbalance* allowed the fixed point approximation of *insertRec* to be more precise.

Unfortunately, the disjunctive form implies larger constraints and longer analysis times. It can also lead to less intelligible constraints for functions with many alternative paths. Our implementation therefore computes the convex hull of constraints at function definitions by default, relying on a programmer directive to explicitly ask for the more precise disjunctive forms when needed.

Stack analysis The stack analysis of the functions of Figure 7.5 yields:

```
lbalance :: {Colour,a,Tree^z1 a,Tree^z2 a}-^z4^z5->Tree^z3 a |
  1+z2=z3, z4=17, z1>=0, z3>=1+z1 ||
  1+z1=z3, z4=17, z2>=0, z3>=1+z2 ||
  2+z2=z3, z4=17, z1>=2, 2+z2>=z1 ||
  z4=17, 1+z1>=z3, z3>=z1, z2>=0, z1>=2, z3>=2+z2
rbalance :: {Colour,a,Tree^z1 a,Tree^z2 a}-^z4^z5->Tree^z3 a |
  1+z2=z3, z4=17, z1>=0, z3>=1+z1 ||
  1+z1=z3, z4=17, z2>=0, z3>=1+z2 ||
  z4=17, 1+z2>=z3, z3>=z2, z1>=0, z2>=2, z3>=2+z1 ||
  2+z1=z3, z4=17, z2>=2, z3>=z2
insertRec :: {Int^z0,Tree^z1 Int}-^z3^z4->Tree^z2 Int |
  z3>=9, 1+9*z1+8*z2>=z3, 1+z1>=z2, 17+9*z1>=z3
insert :: {Int^z0,Tree^z1 Int}-^z3^z4->Tree^z2 Int |
  1+z1>=z2, z3>=8, z2>=1, 9+9*z1+8*z2>=z3, 25+9*z1>=z3
```

The constraints on the latent stack z_4 in *lbalance* and *rbalance* yield a constant cost cost; this is because these functions are not recursive and we are employing a cost lifting transformation which approximates the maximum stack cost among branches.

For the recursive insertion *insertRec* and top level *insert* we get linear bounds that depend on the tree heights z_1 and z_2. Specifically for *insert* we get linear upper bounds for stack: $9+9z_1+8z_2 \geq z_3 \wedge 25+9z_1 \geq z_3$ where z_1 is *input* tree height and z_2 is the *output* tree height (but note that we also have $1+z_1 \geq z_2$ so that the output height is bounded by the input height). The conjunction implies that the stack is bounded by the *minimum* of both expressions: $z_3 \leq \min(9 + 9z_1 + 8z_2, 25 + 9z_1)$.

Heap analysis The heap analysis yields:

```
lbalance :: {Colour,a,Tree^z1 a,Tree^z2 a}-^z4^z5->Tree^z3 a |
  1+z2=z3, z5=18, z1>=0, z3>=1+z1 ||
  1+z1=z3, z5=18, z2>=0, z3>=1+z2 ||
  2+z2=z3, z5=18, z1>=2, 2+z2>=z1 ||
  z5=18, 1+z1>=z3, z3>=z1, z2>=0, z1>=2, z3>=2+z2
rbalance :: {Colour,a,Tree^z1 a,Tree^z2 a}-^z4^z5->Tree^z3 a |
  1+z2=z3, z5=18, z1>=0, z3>=1+z1 ||
  1+z1=z3, z5=18, z2>=0, z3>=1+z2 ||
  z5=18, 1+z2>=z3, z3>=z2, z1>=0, z2>=2, z3>=2+z1 ||
  2+z1=z3, z5=18, z2>=2, z3>=z2
insertRec :: {Int^z0,Tree^z1 Int}-^z3^z4->Tree^z2 Int |
  26*z2+z4>=8+26*z1, 8+26*z1>=z4, z4>=8, 1+z1>=z2, 26*z2+z4>=34
insert :: {Int^z0,Tree^z1 Int}-^z3^z4->Tree^z2 Int |
  14+26*z1>=z4, 1+z1>=z2, z4>=14, 26*z2+z4>=14+26*z1, z2>=1
```

As the stack cost before, the heap cost of *lbalance* and *rbalance* is approximated to the worst-case $z_5 = 18$ which occurs whenever the red-child invariant is violated and the sub-trees need rebalancing. For the top level *insert* we get an upper-bound $14 + 26z_1 \geq z_4$ on the heap cost z_4 as a linear function of the input tree height z_1.

7.2.8 Comparisons with profiling

Table 7.1 presents the results of cost analysis compared with data obtained from an implementation of the Hume abstract machine of Section 6.2. We recall that the Core Hume machine implements unboxed integers and all other values as boxed (including the booleans and empty list constructor). We have implemented an interpreter for the abstract machine that provides exact stack and heap profiling results (i.e. matching the cost model exactly) that can be compared to the analysis results.

The examples are parameterised by the size of data (e.g. list length or tree height). Whenever possible, we chose to generate input data with distribution corresponding to the worst-case costs (e.g. the worst-case for *quicksort* occurs when the input list is ordered). When this was not possible, we approximated the worst-case by choosing the highest cost from some trial runs. Our test entries are:

append-n append two lists of n integers;

nrev-n naive reverse a list of n integers;

reverse-n reverse a list of n integers with accumulating parameter;

reverse-tc-n reverse a list of n integers with accumulating parameter and tail call optimisation;

qsort-n sort an already ordered list of n integers using quicksort;

qsort-d-n destructive version of the above;

msort-n sort a list of n integers using mergesort;

msort-d-n destructive version of the above;

inorder-n in-order traversal of a binary tree with n nodes

ins-tree-n insert an integer into a right-linear binary tree of n nodes (and equal height);

ins-rb-h insert an integer into a red-black balanced tree of height h.

The analysis results for stack and heap are presented as either exact values, upper-bounds or lower-bounds, in order of decreasing precision.

Stack results We obtain upper bounds for the stack cost for all the examples. The upper bounds are exact for reverse (both naive and accumulating version), *quicksort* and in-order tree traversal. We still get good bounds for tree insertion (17% overestimate) and (to a lesser extent) for *mergesort* (43% overestimate).

All the previous examples use cost lifting depth of infinity, i.e. exact cost information. However, to reduce analysis times, we use a coarser cost lifting approximation with depth 2 for the red-black insertion. Consequently, the stack upper bound for this example is slightly less accurate (20% overestimate). Fortunately, we shall see that the precision can be substantially improved by choosing higher cost depth approximations.

Heap results We obtain exact heap results for the accumulating reverse and tree traversal algorithms and good upper bounds for tree insertion (20% overestimate). Of course, no linear upper bounds could be obtained for worst-case quadratic or $O(n \log n)$ algorithms.

For the red-black tree example, we again get somewhat inaccurate bounds (roughly 133% overestimate). This is caused by the analysis taking the conservative assumption that calls to *lbalance* and *rbalance* can always result in red-parent red-child violations (which incur a higher heap cost). Obtaining a more accurate heap bounds on the re-balancing costs would require expressing the global invariant on the tree structure (namely, the red-black tree invariants) which are outside the expressiveness of our size constraints.

7.2.9 Analysis times

Tables 7.2 and 7.3 summarise the analysis time and space for the stack and heap analysis of the previous examples. Timings are discriminated for both top-level and auxiliary definitions. Each entry includes the analysis time (in seconds) and some metrics on the complexity of the constraints before simplification: the number of

	Stack			Heap		
	Profiling	Analysis	Error	Profiling	Analysis	Error
append-1	10	$= 10$	0%	9	$= 9$	0%
append-10	55	$= 55$	0%	72	$= 72$	0%
append-100	505	$= 505$	0%	702	$= 702$	0%
nrev-1	9	≤ 9	0%	5	≥ 5	n/a
nrev-10	63	≤ 63	0%	176	≥ 68	n/a
nrev-100	603	≤ 603	0%	15251	≥ 698	n/a
reverse-1	11	≤ 11	0%	4	$= 4$	0%
reverse-10	56	≤ 56	0%	31	$= 31$	0%
reverse-100	506	≤ 506	0%	301	$= 301$	0%
reverse-tc-1	7	≤ 7	0%	4	$= 4$	0%
reverse-tc-10	7	≤ 7	0%	31	$= 31$	0%
reverse-tc-100	7	≤ 7	0%	301	$= 301$	0%
qsort-100	803	≤ 803	0%	50803	≥ 2290	n/a
qsort-200	1603	≤ 1603	0%	201603	≥ 4590	n/a
qsort-500	4003	≤ 4003	0%	1254003	≥ 11490	n/a
qsort-d-100	803	≤ 803	0%	502	≤ 502	0%
qsort-d-200	1603	≤ 1603	0%	1002	≤ 1002	0%
qsort-d-500	4003	≤ 4003	0%	2502	≤ 2502	0%
msort-100	707	≤ 1001	42%	6553	≥ 2773	n/a
msort-200	1407	≤ 2001	42%	14805	≥ 5573	n/a
msort-500	3507	≤ 5001	43%	42565	≥ 13973	n/a
msort-d-100	708	≤ 1002	42%	412	≤ 651	58%
msort-d-200	1408	≤ 2002	42%	813	≤ 1301	60%
msort-d-500	3508	≤ 5002	43%	2014	≤ 3251	61%
inorder-100	706	≤ 706	0%	301	$= 301$	0%
inorder-200	1406	≤ 1406	0%	601	$= 601$	0%
ins-tree-100	606	≤ 706	17%	506	≤ 606	20%
ins-tree-200	1206	≤ 1406	17%	1006	≤ 1206	20%
*ins-rb-tree-5**	55	≤ 68	24%	49	≤ 114	133%
*ins-rb-tree-10**	95	≤ 113	19%	84	≤ 214	155%
*ins-rb-tree-15**	132	≤ 158	20%	158	≤ 314	99%

Table 7.1: Comparison of the results of profiling and cost analysis.

List functions	time/s	#vars	#eqs	#iters
append	0.02	10	52	5
nrev	0.06	14	113	6
revAcc	0.02	10	52	5
reverse	<0.01	8	24	n/a
insert	0.10	13	266	5
take	0.05	14	124	5
drop	0.04	13	112	5
take'	0.14	15	305	5
drop'	0.12	14	275	5
Total time	0.92			
Total memory	8628 KB			
Quicksort	**time/s**	**#vars**	**#eqs**	**#iters**
split_by	0.24	14	410	5
qsort	0.41	20	338	6
Total time	0.89			
Total memory	15808 KB			
Mergesort	**time/s**	**#vars**	**#eqs**	**#iters**
split	0.12	13	205	5
merge	0.22	14	492	5
msort	2.96	24	887	5
Total time	3.54			
Total memory	50088 KB			

Timings and memory usage obtained on an AMD Athlon XP 2800+ PC, 1 GB of RAM, running Ubuntu GNU/Linux system, kernel 2.6.15, GHC 6.4.2 and PPL 0.9.1.

Table 7.2: Summary of cost analysis times and memory usage.

Binary trees	time/s	#vars	#eqs	#iters
inorder[†]	<0.01	8	24	n/a
inorderAcc[†]	0.05	14	104	5
insert[‡]	2.14	19	3894	5
Total time	2.34			
Total memory	12036 KB			
Red-black trees[*]	**time/s**	**#vars**	**#eqs**	**#iters**
insert[‡]	0.10	10	784	n/a
insertRec[‡]	20.2	24	45840	5
lbalance[‡]	0.54	19	6188	n/a
rbalance[‡]	0.53	19	6188	n/a
Total time	21.3			
Total memory	81660 KB			

Timings and memory usage obtained on an AMD Athlon XP 2800+ PC, 1 GB of RAM, running Ubuntu GNU/Linux system, kernel 2.6.15, GHC 6.4.2 and PPL 0.9.1.

[†] Size measure: the number of inner nodes in the tree.

[‡] Size measure: the maximum height of the tree.

[*] Cost lifting approximation to depth 2.

Table 7.3: Summary of cost analysis times and memory usage (continued).

	Relative error of stack prediction					
	$d = 0$	$d = 1$	$d = 2$	$d = 3$	$d = 4$	$d = 5$
ins-rb-tree-5	75%	60%	55%	55%	31%	7%
ins-rb-tree-10	64%	56%	53%	53%	34%	10%
ins-rb-tree-15	64%	58%	55%	55%	38%	13%
Analysis time (s)	1.9	6.5	9.7	26.2	72.7	97.5
Total memory (KB)	12216	26400	39840	71100	121120	133628

	Relative error of heap prediction					
	$d = 0$	$d = 1$	$d = 2$	$d = 3$	$d = 4$	$d = 5$
ins-rb-tree-5	194%	133%	133%	133%	133%	∞
ins-rb-tree-10	226%	155%	155%	155%	155%	∞
ins-rb-tree-15	156%	99%	99%	99%	99%	∞
Analysis time (s)	2.0	4.5	5.2	9.3	13.7	11.9
Total memory (KB)	12080	17820	21220	30008	34128	29136

Table 7.4: Effect cost lifting depth on analysis time and precision.

variables, the number of (in)equations, and (for recursive functions) the number of iterations required for reaching a fixed-point.[5]

The analysis time is dominated by the polyhedral computations for simplification of size and cost constraints. Therefore, the higher analysis times occur for the more complex algorithms and data structures (e.g. *mergesort* and red-black tree insertion) which generate larger constraints. This can be verified by comparing the number of variables and linear inequations.

We remark that size measure has a large influence on the complexity of generated constraints. For example: the tree height measure introduces a disjunction at each tree node; this leads to an exponential growth of constraint size on nested applications of the tree constructor. This justifies, for example, the constraint complexity and analysis times for the tree-insertion algorithm when compared to tree-traversal: the former uses the tree height (and thus introduces disjunctions), while the later uses the number of nodes (which is expressible as a convex constraint).

[5] Converge of fixed-point iterations is ensured using the widening operator of (Bagnara, Hill, Ricci and Zaffanella 2003) after an initial delay of $k = 3$ iterations.

7.2.10 Effects of the cost lifting transformations

For most of the examples in Tables 7.1 and 7.2 we have conducted the most precise analysis possible within our framework. However, for the red-black tree insertion algorithm we have employed the cost lifting approximation described in Section 6.4.6 to reduce the complexity of synthesised constraints.

We now address the issue of whether we can effectively improve the analysis precision by using a higher depth, and what impact does that have on analysis time. Table 7.4 summarises the results of increasingly large depths d in the stack and heap analysis of the red-black tree insertion. The precision is measured as the relative error between predicted and measured costs. The values for depth $d = 0$ correspond the results of Table 7.1.

Choosing depth $d = 5$ greatly improves the precision of the stack upper bound to just 13% overestimation of the measured stack cost. This is at the expense of (roughly) a 50-times increase in analysis time. Still, we believe that a few minutes of static analysis is an acceptable price to pay for an accurate, guaranteed worst-case stack prediction (particularly since the higher precision analysis can be employed only when required).

For the heap analysis, depths higher than $d = 1$ yield no further improvements. In fact, choosing $d = 5$ (or indeed any larger value) yields *worst* heap results: ∞ indicates that the analysis failed to obtain *any* upper bound.

This loss of precision is caused by application of the widening operator during the fixpoint approximation of *insertRec*. We can verify this by examining the constraints for the first three iterations (i.e. before any application of the widening operator). For brevity, we omit inequations that do not express an upper bound on the heap cost z_4.

$$
\begin{aligned}
&\phi_0 : \mathit{False} \\
&\phi_1 : 6z_1 + z_4 = 2 + 6z_2 \wedge \ldots \\
&\phi_2 : 28 + 6z_2 \geq 6z_1 + z_4 \wedge 24 + 6z_1 + z_4 \wedge 28 \geq z4 \wedge 8 + 10z_1 \geq z_4 \wedge \ldots \\
&\phi_3 : 54 + 6z_2 \geq 6z_1 + z_4 \wedge 58z_2 \geq 50 + 6z_1 + z_4 \wedge 48 \geq z_4 \wedge \\
&\qquad 8 + 15z_1 \geq z_4 \wedge 18 + 10z_1 \geq z_4 \wedge 10z_1 + 10z_2 \geq 2 + z_4 \wedge \ldots
\end{aligned}
$$

Approximations ϕ_2 and ϕ_3 express bounds $8+10z_1 \geq z_4$ and $8+15z_1 \geq z_4$ relating z_4 to the size z_1. We might expect that some bound of the form $8+cz_1 \geq z_4$ will hold for fixpoint; indeed, for $c = 20$ the bound will be a sound approximation. However, the fixpoint iterations will approximate it strictly from bellow (i.e. generating constraints with coefficients $10 < c < 20$), which means that the constraints are unstable across iterations. To ensure convergence, the widening operator is used at some point and discards all unstable bounds, which means that no safe heap approximation is found.

When doing the approximation for *insertRec* with a cost lifting depth $d < 5$ there is some overestimation in the coefficients of cost annotations. This results in reaching the stable bound $8 + 20z_1 \geq z_4$ after only two iterations. The widening operator will then not discard this bound because it is stable. Therefore, the coarser cost annotation results in some extrapolation that allows the widening to reach a better fixpoint approximation.

While we do not attempt to solve this problem, we remark that choosing different cost depth allowed our analysis to avoid it in all (few) situations where we encountered it.

Finally, we remark that although we have presented our analysis as one type and effect system, the results of Table 7.4 indicate that we might obtain better results by running the stack and heap analyses separately (so that we can employ different depths for cost lifting).

7.3 Embedded systems

In this section we present some simplified embedded systems implemented as Hume processes that respond to external inputs, maintain some internal state and produce outputs. Such systems are typically intended to run indefinitely and are thus required to have bounded space behaviour.

We chose problems that benefit from the expressiveness of the Core Hume language (e.g. recursive data types and function definitions) and employ our cost analysis to obtain bounded space guarantees. Finally, we assess the accuracy of the analysis by comparing the static bounds obtained with information obtained from profiling.

7.3.1 Mine pump controller

Our first example is a simplified mine pump controller that has been presented as a case-study in the design of real-time embedded systems; other languages solutions include ADA with real-time extensions (Burns and Wellings 1996), SparkADA (SPARK Team 2005) and finite state Hume (Hammond and Michaelson 2002).

The software controls a pump that drains water accumulating in a mine shaft to the surface. The pump operates in either automatic or manual mode. In automatic

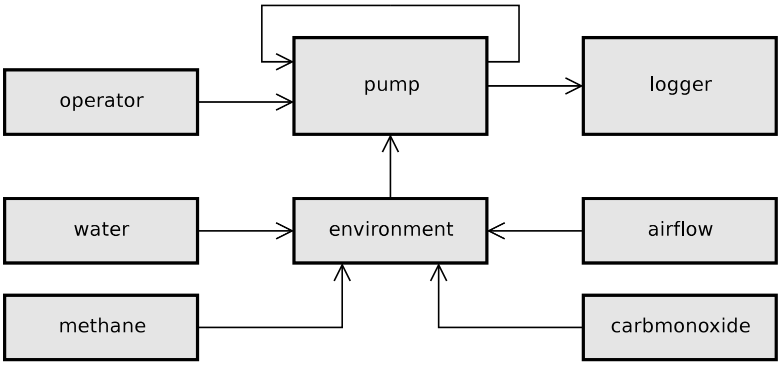

Figure 7.6: Diagram of box wiring for the mine pump controller.

mode, the pump turns on or off when the water level in the mine shaft reaches high or low marks (indicated by two sensors in the shaft). In manual mode the pump responds to operator requests.

Apart from the pump control itself, the system should also monitor environment readings for methane and carbon monoxide gases and the flow of air in the mine; these should be reported to a logging system. The main safety requirement is that the pump must not be turned on when the methane level is too high to avoid the risk of explosion; this must be enforced in both automatic and manual operation.

Our Core Hume implementation does not employ recursive data types or recursive functions; this means that stack and heap bounds could be obtained by a simpler static analysis such as the one by Hammond and Michaelson (2002). We chose to apply our cost analysis to this simpler example to demonstrate that we still get tight space bounds for non-recursive programs and that the analysis is practical in such cases.

Figure 7.6 depicts the network of Hume boxes implementing the system processes. The main reactive components are the pump control (*pump*) and environment monitor (*environment*). In order to run the mine controller and collect profiling data, the system also includes several processes providing simulated inputs: boxes *water*, *methane*, *carbmonoxide* and *airflow* are simulators of the physical sensors; the *operator* box generates pseudo-random asynchronous requests from a human operator; finally, the *logger* box simulates a console for event logging.

In the following sections we will describe the main component, namely the *pump* box. For the complete listing of mine pump system in Core Hume, see Appendix B.1.

Data types

We first define the data types for representing the pump state (on or off), the operation mode (automatic or manual) and the requests from the operator (turn the pump on or off, switch to automatic or manual mode).

```
data STATE = ON | OFF ;       -- pump state

data MODE = AUTO | MANUAL ;   -- pump operating mode

-- operator requests
data REQ = REQ_ON | REQ_OFF | REQ_AUTO | REQ_MANUAL ;
```

We also define a type synonym for the readings from the low and high level watermark sensors.

```
type WATER = (Bool,Bool)    -- (low,high)
```

Pump process

The inputs to the pump box are: the current state and mode, the water sensor readings, the methane alarm; the outputs are the new state and mode and a log string.

```
box pump
in (mode::MODE, state::STATE, water_sensors::WATER,
     ch4_alarm::Bool, req::REQ)
out (mode'::MODE, state'::STATE, log::String)
```

The rule matching is unfair to be able to prioritise the inputs: the first rule deals with the most critical response, i.e. turning the pump off if the methane alarm input is `True`:

```
unfair
 (mode, ON, water, True, *) ->
         (copyMODE mode, OFF, "CH4 alarm: turning pump off")
```

In this rule and others we use functions `copyMODE` and `copySTATE` on the right-hand side to copy values of the corresponding data types onto output wires.[6] Note also

[6] Since the data types are non-recursive, we could alternatively expand the box rules to cover all possible cases. However, this would in general lead to an exponential blow-up in the number of rules, so would be impractical for data types with a large number of constructors. The use of copying functions does not suffer from this scalability issue.

the use of a *-pattern to ignore (but not consume) the operator request input.

The next rules deal with responses when the pump is in automatic mode.

```
| (AUTO, state, water, alarm, REQ_MANUAL) ->
      (MANUAL, copySTATE state, "Operator requested MANUAL mode")
| (AUTO, state, water, False, _*) ->
      (AUTO, auto_control state water, "Running in auto mode")
```

The first of these rules acknowledges requests to switch to manual mode; the second rule turns the pump on or off according to the water level sensors (implemented in function auto_control). Note that the second rule only applies if the methane alarm is False; this prevents turning on the pump in an unsafe situation. Also, note the use of the _*-pattern to ignore and consume any other operator requests while in automatic mode.

The remaining rules deal with responses in manual mode.

```
| (MANUAL, state, water, False, REQ_ON) ->
      (MANUAL, ON, "Operator requested pump ON")
| (MANUAL, state, water, alarm, REQ_OFF) ->
      (MANUAL, OFF, "Operator requested pump OFF")
| (MANUAL, state, water, alarm, REQ_AUTO) ->
      (AUTO, copySTATE state, "Operator requested AUTO mode")
  -- default rule:
| (mode, state, alarm, _*) -> (copyMODE mode, copySTATE state, *)
```

The first rule deals with operator requests to turn the pump on; again, it only applies when the methane alarm is False. The final rule is a default that maintains the current pump state and mode in all remaining situations. Note the use of a *-expression on the right-hand side of the rule to indicate that no message should be sent to the logger.

Results of size and cost analysis

Table 7.5 summarises the results obtained for cost analysis of the mine pump controller. The results are discriminated for each wire identified by an output port. The analysis results are the upper-bound reported by our static analysis. The profiling data is the worst-case collected from running 10^4 scheduling iterations on the Core Hume abstract machine interpreter. Since the simulation boxes generate random inputs, no external input data was used.

	Stack		Heap	
Wire	Profile	Analysis	Profile	Analysis
airflow.level	3	3	14	14
airflow.time'	3	3	4	4
carbmonoxide.level	3	3	14	14
carbmonoxide.time'	3	3	4	4
environment.airflow_alarm	5	5	3	3
environment.ch4_alarm	5	5	3	3
environment.ch4_alarm'	5	5	3	3
environment.co_alarm	5	5	3	3
logger.dummy	2	2	29	29
methane.level	3	3	14	14
methane.time'	3	3	4	4
operator.time'	3	3	0	0
operator.trigger	4	4	1	1
pump.log	5	5	31	31
pump.mode'	7	7	1	1
pump.state'	8	8	1	1
request.req	5	5	5	5
sink.dummy'	1	1	0	0
water.sensors	5	5	19	19
water.time'	3	3	4	4
Total (maximum/sum)	8	8	157	157

Table 7.5: Results of cost analysis for the mine pump controller.

As might be expected since the mine pump is a finite-state machine, the upper bounds obtained by the analysis match the worst-case profiling results exactly for both stack and heap in all wires. This demonstrates that we can still obtain good results from the type and effect analysis when applied to finite-state Hume programs.

7.3.2 Lifts controller

The second example is based on a Hume example by Greg Michaelson. The program simulates a two-lift controller for a building with several floors (a parameter of the problem). Passengers request a lift by pressing a button on the landing. The controller should dispatch lifts to floors, minimising the number of stops and the movement range for each lift. The complete listing for this example is included in Appendix B.2.

Representation of lift states

We represent floors by integer numbers starting from zero. The pending requests for a lift are represented by a list of booleans.[7] If the i-th element is True then the lift must wait at floor i; initially, the list consists only of False elements, i.e. there are no pending requests.

A lift can be moving, stopped or waiting for passengers; this state is represented by a sum type Move. The complete state of a lift is a 5-tuple consisting of the movement direction, the current floor, the lowest and highest request and the list of pending requests. The corresponding Core Hume type declarations are:

$$\begin{aligned}
\mathsf{type\ Floor} &= \mathsf{Int} \\
\mathsf{type\ Pending} &= \mathsf{List\ Bool} \\
\mathsf{data\ Move} &= \mathsf{UP} \mid \mathsf{WAIT\_UP} \mid \mathsf{DOWN} \mid \mathsf{WAIT\_DOWN} \mid \mathsf{STOP} \\
\mathsf{type\ State} &= (\mathsf{Move}, \mathsf{Floor}, \mathsf{Floor}, \mathsf{Floor}, \mathsf{Pending})
\end{aligned}$$

Changes to lift states are done by two functions with type signatures

$$\begin{aligned}
\mathit{req\_lift} &:: \{\mathsf{State}, \mathsf{Floor}\} \to \mathsf{State} \\
\mathit{move\_lift} &:: \mathsf{State} \to \mathsf{State}
\end{aligned}$$

that, respectively, add a new request and move the lift (if it is not stopped). Both

[7] Alternatively, we could extend Core Hume with a bit-vector primitive type with constant-cost access. We do not pursue this optimisation here because it would not invalidate the results obtained, except presumably by lowering the overall costs by some absolute constant.

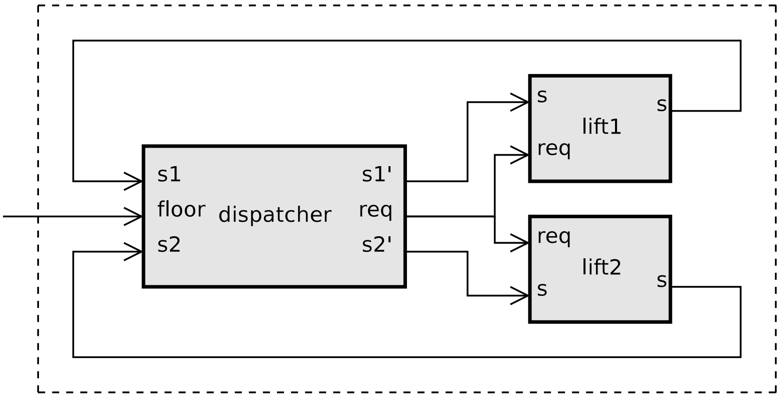

Figure 7.7: Diagram of box wiring for the two-lifts controller.

functions maintain the invariant that for any state $(s, floor, lo, hi, pend)$,

$$s \neq \mathsf{STOP} \implies (\forall i.\quad 0 \leq i < floors \wedge pend[i] = \mathsf{True} \implies lo \leq i \leq hi)$$

i.e. if the lift is moving then *lo* and *hi* are lower and upper bounds for the pending requests (where *floors* is the number of floors and *pend*[*i*] is the *i*-th element of the list *pend*).

Finally, we remark that *req_lift* and *move_lift* preserve the length of the list of pending requests. This invariant will be inferred by the size analysis and is essential in guaranteeing that the State data type has bounded heap cost.

Dispatching lifts to floors

The coordination of the lifts controller is implemented using one box for each lift and one box for a dispatcher that splits requests across lifts. The only external input to the controller is a port to request a lift.[8] Each lift box maintains a representation of the internal state wired as a feedback loop. For simplicity, there are no outputs from the lifts (e.g. to command an engine). Figure 7.7 depicts the communication wires between boxes.

To minimise the total distance travelled, each lift serves requests in a single direction before reversing. To select which lift should serve a request, the dispatcher computes the "distance" (a positive number of floors) that each lift must travel to reach the requested floor, taking in account not just the current location but also the destination and direction of the lift. This computation is encapsulated in the

[8] Since there is no mechanism for hiding port visibility in Core Hume, the external/internal distinction is merely illustrative of the intended use.

function

$$distance :: \{\mathsf{State}, \mathsf{Floor}\} \to \mathsf{Int}$$

defined by cases analysis on the lift state. For example, the case for the lift moving up is:

$$\begin{aligned} & distance\ (\mathsf{UP}, floor, lo, hi, pend)\ dest \\ & \qquad = \begin{cases} dest - floor & \text{if } dest \geq floor \\ 2(hi - floor) + (floor - dest) & \text{if } dest < floor \end{cases} \end{aligned}$$

The expression $2(hi - floor) + (floor - dest)$ is the round-trip distance from $floor$ to hi plus the distance from $floor$ to $dest$. Simplifying this expression, we get

$$distance\ (\mathsf{UP}, floor, lo, hi, pend)\ dest = \begin{cases} dest - floor & \text{if } dest \geq floor \\ 2hi - floor - dest & \text{if } dest < floor \end{cases}$$

The remaining cases are similar; we refer the reader to Appendix B.2 for the complete listing.

Applying size and cost analysis

We can directly apply the cost analysis to the data structures and functions used in the lift controller. The user-defined data type Move is unsized; integers and lists have the default sizes.

The analysis obtains good size and cost bounds automatically for most of the functions. The exceptions are *req_lift* and *move_lift* where the default analysis obtains lower bounds for stack and heap costs, but no upper bounds. This is caused by the use of the *convex hull* to simplify constraints: *req_lift* and *move_lift* are defined by disjoint cases, some of which have constant costs while other have linear costs; computing the convex hull across such disjunctions loses the upper bounds.

We can retain cost upper bounds by specifying an “no-hull” directive for the analysis of the two functions. Although not hulling avoids the loss of precision, it also results in a much larger constraint in disjunctive form.[9] For example, the

[9] We recall that our constraint solver computes the disjunctive form to employ operations for systems of convex linear constraints (see Section 5.5).

	Stack		Heap	
Wire	Profile	Analysis	Profile	Analysis
dispatcher.s1	75	75	106	106
dispatcher.s1'	54	57	48	48
dispatcher.s2	75	75	106	106
dispatcher.s2'	54	57	48	48
dispatcher.req	16	16	5	5
Total (maximum/sum)	75	75	313	313

Table 7.6: Results of cost analysis for the lift controller.

analysis of *req_lift* yeilds the annotated type and constraint

$$\{(\mathsf{Move},\ \mathsf{Int}^{z_0},\ \mathsf{Int}^{z_1},\ \mathsf{Int}^{z_2},\ \mathsf{List}^{z_3}\ \mathsf{Bool}), \mathsf{Int}^{z_4}\} \xrightarrow{z_9;z_{10}} (\mathsf{Move},\ \mathsf{Int}^{z_5},\ \mathsf{Int}^{z_6},\ \mathsf{Int}^{z_7},\ \mathsf{List}^{z_8}\ \mathsf{Bool})$$

$$\begin{aligned}
&(z_3 = z_8 \wedge 13 + 4z_4 = z_{10} \wedge z_9 = 4 \wedge z_0 = z_5 \wedge z_4 \geq 0 \wedge \\
&\quad z_1 \geq z_6 \wedge z_7 \geq z_4 \wedge z_8 \geq 1 + z_4 \wedge z_7 \geq z_2 \wedge z_4 \geq z_6) \vee \\
&(z_3 = z_8 \wedge 13 + 4z_4 = z_{10} \wedge z_9 =_9 \wedge z_0 = z_5 \wedge z_4 \geq 0 \wedge \\
&\quad z_1 \geq z_6 \wedge z_7 \geq z_4 \wedge z_8 \geq 1 + z_4 \wedge z_7 \geq z_2 \wedge z_4 \geq z_6) \ \vee \cdots
\end{aligned}$$

where we have omitted the remaining 22 disjunctions. We could simplify the result for presentation purposes by factoring common inequations (e.g. the list size invariant $z_3 = z_8$ that holds in every disjunction). However, we are ultimately interested in results of the coordination layer analysis which are always simple intervals for the sizes and costs of wires. For this purpose the disjunctive form is more convenient.

Results of size and cost analysis

We ran the analysis and profiling for a simulation of a ten floor building. The coordination level analysis obtains the following annotated type

$$(\mathsf{Move},\ \mathsf{Int}^{[0,\,9]},\ \mathsf{Int}^{[0,\,9]},\ \mathsf{Int}^{[0,\,9]},\ \mathsf{List}^{[10,\,10]}\ \mathsf{Bool})$$

for all wires carrying lift states, namely *dispatcher.s1*, *dispatcher.s1'*, *dispatcher.s2* and *dispatcher.s2'*. The only remaining wire *dispatcher.req* has type Request with no size information. From this we can verify that the lists of pending requests have constant length 10 and that floor indices are always between 0 and 9.

Table 7.6 presents the corresponding results for cost analysis for the lift-controller. Each line corresponds to one wire, identified by a dispatcher port. The profiling data was obtained by running 10^4 scheduler iterations with random requests.

The analysis obtains the exact stack and heap cost for most of the wires. There is only a very slight overestimation for the stack cost of the output *dispatcher.s1'* and *dispatcher.s2'*. The heap analysis matches the worst-case data from profiling exactly.

The total results reflects the space requirements for the lift controller as a whole: since the heap for each wire is separate, the total heap required is proportional to the sum of the wire heaps.[10] A single stack can be shared among wires[11] and therefore its size is the maximum of each wire. No extra dynamic memory is required for the Core Hume machine, so we have obtained a tight prediction that guarantees bounded space behaviour.

7.3.3 Geometric region server

Our third example is inspired by a comparison between Haskell and other languages in an experiment on software prototyping described by Hudak and Jones (1994). The experiment was conducted by the Naval Surface Warfare Center (NSWC) and consisted of giving ten volunteer programmers an informal specification of a "geometric region server" (geo-server). Each participant submitted a working prototype in specific programming language together with documentation and a record of development time. Hudak and Jones submitted an entry in the Haskell programming language. All entries were independently assessed according to various software development metrics.

The geo-server is a simplified component of a real NSWC system. The inputs of the geo-server are positions of friend and foe "objects" (i.e. ships or aircraft); around some of these there are geometric zones of different shapes; the geo-server should detect and report the intersections of objects with these zones. The complete Core Hume listing for this example is included in Appendix B.3.

Representing geometric regions

The Haskell solution described by Hudak and Jones represents a two-dimensional region as a function from points to booleans:

$$\mathsf{type\ Point} = (\mathsf{Float}, \mathsf{Float})$$

$$\mathsf{type\ Region} = \mathsf{Point} \to \mathsf{Bool}$$

[10] Since each wire is associated with *two* heap regions of equal size (see Section 6.2), the total heap footprint is *twice* the sum of the wire heaps.

[11] Because the scheduler is non-preemptive.

Testing for region membership is simply function application:

$$\begin{aligned}&\mathit{inRegion} :: \mathsf{Point} \rightarrow \mathsf{Region} \rightarrow \mathsf{Bool}\\ &\mathit{inRegion}\ p\ r \ = \ r\ p\end{aligned}$$

Simple regions can be defined directly, e.g. the region enclosed by a circle with a given centre and radius:

$$\begin{aligned}&\mathit{circle} :: \mathsf{Point} \rightarrow \mathsf{Float} \rightarrow \mathsf{Region}\\ &\mathit{circle}\ (x_0, y_0)\ r \ = \ \lambda(x, y) \rightarrow \ \mathsf{let}\ \ dx = x - x_0\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad dy = y - y_0\\ &\qquad\qquad\qquad\qquad\qquad\quad\ \mathsf{in}\ \ dx * dx + dy * dy \leq r * r\end{aligned}$$

Complex regions can be obtained by application of *region combinators* to simpler ones. For example, the function

$$\begin{aligned}&\mathit{annulus} :: \mathsf{Point} \rightarrow \mathsf{Float} \rightarrow \mathsf{Float} \rightarrow \mathsf{Region}\\ &\mathit{annulus}\ p\ r_1\ r_2 \ = \ \mathit{intersect}\ (\mathit{circle}\ p\ r_2)\ (\mathit{outside}\ (\mathit{circle}\ p\ r_1))\end{aligned}$$

defines an annular region using circles and combinators for the *outside* of a region and the *intersection* of regions:

$$\begin{aligned}&\mathit{outside} :: \mathsf{Region} \rightarrow \mathsf{Region}\\ &\mathit{intersect} :: \mathsf{Region} \rightarrow \mathsf{Region} \rightarrow \mathsf{Region}\end{aligned}$$

It is straightforward to define combinators like these and to add more basic shapes.

This "domain specific language" for geometric zones was one of the reasons why the Haskell prototype was more concise and readable than the Ada and C++ solutions, and so we would like to transpose it to our Hume program. However, we cannot express higher-order combinators like *outside* and *intersect* in a first-order language such as Core Hume. Instead, we first employ the standard defunctionalisation program transformation (Reynolds 1972, Danvy and Nielsen 2001) to replace the higher-order representation by a first-order data type. Our data type for regions is as follows (including half-planes shapes and a union operation):

$$\begin{aligned}\mathsf{data\ Region} = \ &\mathsf{Above\ Float} \mid \mathsf{Below\ Float} \mid \mathsf{Right\ Float} \mid \mathsf{Left\ Float}\\ \mid\ &\mathsf{Circle\ Point\ Float} \mid \mathsf{Outside\ Region}\\ \mid\ &\mathsf{Intersect\ Region\ Region} \mid \mathsf{Union\ Region\ Region}\end{aligned}$$

The region membership test function is the "apply" function resulting from defunctionalisation:

$$\begin{aligned}
&\mathit{inRegion} :: \{\mathsf{Point}, \mathsf{Region}\} \to \mathsf{Bool}\\
&\mathit{inRegion}\ (x,y)\ (\mathsf{Above}\ y_0)\ =\ y_0 \le y\\
&\mathit{inRegion}\ (x,y)\ (\mathsf{Below}\ y_0)\ =\ y \le y_0\\
&\mathit{inRegion}\ (x,y)\ (\mathsf{Right}\ x_0)\ =\ x_0 \le x\\
&\mathit{inRegion}\ (x,y)\ (\mathsf{Left}\ x_0)\ =\ x \le x_0\\
&\mathit{inRegion}\ (x,y)\ (\mathsf{Circle}\ (x_0,y_0)\ r)\ =\ \begin{array}{ll}\mathsf{let} & dx = x - x_0\\ & dy = y - y_0\\ \mathsf{in} & dx * dx + dy * dy \le r * r\end{array}\\
&\mathit{inRegion}\ p\ (\mathsf{Outside}\ r)\ =\ \mathit{not}\ (\mathit{inRegion}\ p\ r)\\
&\mathit{inRegion}\ p\ (\mathsf{Intersect}\ r_1\ r_2)\ =\ \mathsf{if}\ \mathit{inRegion}\ p\ r_1\ \mathsf{then}\ \mathit{inRegion}\ p\ r_2\ \mathsf{else\ False}\\
&\mathit{inRegion}\ p\ (\mathsf{Union}\ r_1\ r_2)\ =\ \mathsf{if}\ \mathit{inRegion}\ p\ r_1\ \mathsf{then\ True\ else}\ \mathit{inRegion}\ p\ r_2
\end{aligned}$$

The function *inRegion* is now first-order and recursive, so we can apply our cost analysis to obtain stack and heap bounds. All that remains to be done is to assign a size measure to the region data type. It is immediate that, in the worst-case, *inRegion* will perform a complete traversal of the data type; therefore, we define the size measure for regions to be the total number of inner nodes:

$$\begin{aligned}
\mathsf{data\ Region}^n = {} & \mathsf{Above\ Float} \quad \{n = 0\}\\
& \vdots \qquad \text{(size zero for other shapes)}\\
\mid\ & \mathsf{Circle\ Point\ Radius} \quad \{n = 0\}\\
\mid\ & \mathsf{Outside\ Region}^p \quad \{0 \le p \wedge n = 1 + p\}\\
\mid\ & \mathsf{Intersect\ Region}^p\ \mathsf{Region}^q \quad \{0 \le p \wedge 0 \le q \wedge n = 1 + p + q\}\\
\mid\ & \mathsf{Union\ Region}^p\ \mathsf{Region}^q \quad \{0 \le p \wedge 0 \le q \wedge n = 1 + p + q\}
\end{aligned} \tag{7.9}$$

With this data declaration our analysis obtains the following size and cost bounds:

```
inRegion :: {(Float,Float),Region^z0}-^z2^z3->Bool^z1 |
  1>=z1, z1>=0, z0>=0, 9*z3>=1+z2, 89+101*z0+7*z3>=12*z2, z3>=1,
  z2>=4,  15+9*z0>=z2, 13+13*z0>=z3
```

Among other relations, we obtain a stack upper bound $15 + 9z_0 \ge z_2$ and a heap upper bound $13 + 13z_0 \ge z_3$ as functions of the region size z_0.

Implementation of the geo-server

The geo-server itself is implemented as a Hume box with two inputs: the position of a "friendly" object (called *host*) and a "foe" object (called *target*). The geo-server

		Stack analysis			
Wire	Profile	default ($c = 0$)	$c = 3$	$c = 6$	$c = 9$
geoserver.weapon	26	26	46	73	100
geoserver.tight	28	44	37	37	37
geoserver.engage	34	35	82	136	190
Total (maximum)	34	44	82	136	190

		Heap analysis			
Wire	Profile	default ($c = 0$)	$c = 3$	$c = 6$	$c = 9$
geoserver.weapon	34	46	36	34	34
geoserver.tight	35	83	48	38	36
geoserver.engage	49	61	51	49	49
Total (sum)	118	190	135	121	119

Table 7.7: Results of cost analysis for the geo-server.

tracks three regions of interest:

1. an “engage-ability zone” with an annular shape centred on the host;
2. a “weapons doctrine zone” with a semi-circular shape centred on the host;
3. a “tight zone” of rectangular shape on fixed coordinates.

The outputs of the geo-server are three boolean values representing the intersection of the target with each of these zones.

Results of cost analysis

Table 7.7 presents the results of cost analysis for the geo-server box. Each line corresponds to one of the output wires. The first column contains profiling data obtained from running 10^4 scheduler iterations with random input points for the host and target positions. The remaining columns contain the upper-bounds obtained with different values of a parameter (described ahead). We also include the results for the maximum stack depth and total heap allocated for the three wires; these accumulated results can be directly compared for precision across the different analyses.

The “default” column reports results with the region data type as in (7.9). We get relatively good stack upper bounds: there is a 30% over-estimative on the tight-zone wire but the remaining two are (almost) exact. The default analysis for heap yields worse results: over 100% for the tight-zone wire and 61% over-estimative in total.

The cause of heap over-estimation is our uniform choice of size for basic regions. Inspecting the definition of *inRegion* we can see that the case of a circle incurs a higher heap cost than other basic regions (i.e. the half-planes Above, Below, Left and Right). However, since all basic regions are assigned the same size of zero, the analysis over-approximates the costs of each of these to the worst-case.

We can attempt to improve precision by simply choosing the size measure of a circle to be larger than that of a half-plane. The optimum value will depend on the precise relation between costs of each shape (and, in fact, will be different for stack and heap). However, since the soundness of analysis is not dependent on the size measure, we can simply re-run the analysis with different size type declarations and choose the best (lowest) prediction.

Table 7.7 includes also the results obtained with different sizes for the circle shape (and all other basic shape with size zero). The best results for heap (in fact: almost exact) were obtained for column marked $c = 9$ corresponds to running the analysis with the sized type declaration:

$$\begin{array}{rl}
\mathsf{data\ Region}^n = & \mathsf{Above\ Float} \quad \{n = 0\} \\
& \vdots \\
& \mid \mathsf{Circle\ Point\ Radius} \quad \{n = 9\} \\
& \mid \mathsf{Outside\ Region}^p \quad \{0 \leq p \wedge n = 1 + p\} \\
& \mid \mathsf{Intersect\ Region}^p\ \mathsf{Region}^q \quad \{0 \leq p \wedge 0 \leq q \wedge n = 1 + p + q\}
\end{array}$$

However, the stack prediction using the above declaration is *worse* than with uniform sizes. This is because, unlike the heap costs, the stack costs of each shape are similar. The example suggests that, in general, better cost predictions might be possible by conducting separate stack and heap analyses with different size declarations.[12]

[12] In Section 7.2.10 we made the same remark regarding the choice of cost annotation depth.

Chapter 8

Conclusion

In this chapter we review our contributions, compare our approach with previous research in automatic cost analysis, highlight the limitations of our approach and point some directions for further research.

8.1 Summary

The research motivation for this thesis was to provide guaranteed time and space bounds for functional programs in resource-sensitive systems; we chose the functional language *Hume* that specifically targets embedded and real-time systems as a basis for this research. Since resource bounds are incomputable for programming languages of sufficient expressiveness (which full Hume certainly is), we set out to explore a partial solution by restricting the programming language and by employing a source-level analysis that infers approximate cost bounds and may yield uninformative answers.

Our aim was, therefore, only partially achieved: we considered a core subset of Hume with first-order recursive functions and data structures but not higher-order functions, and an analysis that predicts space but not time costs. Moreover, to obtain an automatic analysis, we have restricted our attention to inferring space bounds expressed as linear inequations. Nonetheless, our cost analysis is capable of automatically obtaining sound and accurate worst-case bounds of dynamic stack and heap costs for functional programs with lists, binary trees and other recursive data structures and for simplified embedded systems using these.

In this chapter we review our contributions, the limitations of our approach and highlight topics for further research.

8.2 Contributions

We now review the contributions made in this thesis, in order of relevance.

8.2.1 A static analysis for size and space costs

In Chapter 6 we presented an *automatic static analysis for bounding sizes, stack and heap costs* of Core Hume, a subset of the Hume language with first-order recursive functions and recursive data types. Our analysis combines ideas from previous works on sized type systems (Hughes et al. 1996, Chin and Khoo 2001), type and effect systems for costs (Dornic et al. 1992, Reistad and Gifford 1994, Hughes and Pareto 1999) and cost analysis of Hume programs (Hammond and Michaelson 2002). It extends previous approaches in three main directions:

1. previous sized type systems focused on natural numbers, lists and streams (Reistad and Gifford 1994, Hughes et al. 1996, Chin and Khoo 2001); our extension for user-defined sizes allows the application of sized types to non-linear data structures e.g. binary trees;

2. the use of abstract interpretation techniques to automatically infer cost bounds of recursive functions is also novel[1] and avoids the need for insightful user-supplied stack and heap bounds in type annotations required in previous works (Hughes and Pareto 1999, Pareto 2000);

3. our size and cost analysis for Core Hume extends previous work for a finite-state subset of Hume (Hammond and Michaelson 2002) and demonstrates that it is possible to extend the guarantees on bounded space behaviour for Hume programs employing recursive data types and functions.

Static analyses can be evaluated according to the range of language features covered, the precision of information obtained and the efficiency of the analysis. In general, improving some of these aspects involves a trade-off: a fast analysis might yield less precise results; and dealing with more language features can increase analysis time or reduce precision of simpler cases.

Assessing a cost analysis for a subset of a research language is particularly difficult, since case-studies must be hand-adapted or made up from scratch (as were the examples of Chapter 7). Direct quantitative comparisons with other approaches are also not straightforward because of differences in languages and cost models.

[1] The use of widening and hulling for automatically inferring *sizes* (but not cost bounds) was proposed by Chin et al. (2003).

We instead resort to a qualitative comparison with the most directly related works, namely the stack and heap analysis for Embedded ML of Hughes and Pareto (1999) and the amortised heap analysis of Hofmann and Jost (2003):

1. *Formal basis.* Our analysis is formally specified in Section 6.4 as a type and effect system; it is proved correct against a cost-instrumented semantics in Section 6.5; the model of costs is specified in Section 6.2 as an abstract machine. Both other approaches (Hughes and Pareto 1999, Hofmann and Jost 2003) are also formally based.

2. *Cost model.* Our model of stack and heap costs is based on an abstract machine that mimics a realistic implementation e.g. by accounting stack costs of call frames and pattern matching, and heap costs for tags. We therefore argue that the cost bounds can be transposed to a concrete realisation of the abstract machine by a simple choice of units. Moreover, we have shown in Section 6.6 that both cost model and analysis can be extended to account for common compiler optimisations. Our abstract machine is inspired by the one of Hughes and Pareto which also accounts for stack and heap; the cost model of Hofmann and Jost, however, accounts only for heap.

3. *Automation.* Our analysis can infer size and cost inequations for recursive and non-recursive functions without requiring user intervention; in particular, it does *not* require insightful cost annotations as does the analysis of Hughes and Pareto. Like our analysis, however, the amortised analysis of Hofmann and Jost is also automatic.

4. *Modularity.* Like other program analysis based on type and effect systems, our cost analysis is modular, that is, it can infer cost information separate from its uses; it is thus possible to perform separate analysis of libraries or modules. Although we have not demonstrated it in this thesis, we believe that this modularity is an important characteristic for ensuring the scalability of the analysis to large programs. This should also apply to the approaches taken by Hughes and Pareto and Hofmann and Jost.

5. *Intelligibility.* The bounds obtained by our cost analysis can be directly related to the source-code as augmented type signatures for functions; this is not so straightforward e.g. in the case of analysis of machine code blocks (Ferdinand et al. 2001). Moreover, it is possible to employ automatic simplifications to report partial information (e.g. focus just on sizes, stack or heap costs). These remarks should equally apply to the approaches of Hughes and Pareto and Hofmann and Jost.

6. *Accuracy.* The experimental results reported in Table 7.1 of Section 7.2 demonstrate that our analysis obtained *exact* worst-case costs for simple programs, but was still able to obtain finite bounds (albeit less accurate) for more complex ones e.g. the list sorting and tree insertion algorithms. The analysis of Hughes and Pareto is limited to linear data structures and yields poor results with irregular divide-and-conquer algorithms (e.g. quicksort); the analysis of Hofmann and Jost deals with these naturally by simply splitting the potential between uses.

7. *Trade-off between precision and time.* Our analysis allows a selective trade-off between the precision of cost bounds and analysis time by using known abstract interpretation techniques (e.g. delaying the use of widening or hulling) and more importantly, by the *cost lifting* transformation of Section 6.4.6 that allows moving cost annotations from inner to enclosing expressions. The effectiveness of this heuristic was demonstrated in some examples e.g. red-black tree insertion of Section 7.2.7. Moreover, cost lifting can be limited to specific functions so that the precision loss is localised. Neither Hughes and Pareto nor Hofmann and Jost have considered this issue.

8. *Usability in a compiler.* In Section 6.7 we have demonstrated that the bounds obtained by our type and effect analysis for Core Hume expressions can be automatically used in a compiler to obtain memory bounds for communication wire of the coordination layer. The size and cost annotations of Hughes and Pareto specify static sizes for memory regions which could, in principle, also be used in a compiler; the analysis of Hofmann and Jost, however, obtains heap costs as functions of unknown input and output sizes and therefore is not usable in a compiler unless extra size information is provided.

9. *Efficiency.* Timing results from our prototype implementation reported in Tables 7.2 and 7.3 show that the analysis fits an average desktop computer and that the analysis time is acceptable for a verification phase: our largest example (red-black tree insertion) took 20 seconds to analyse.

   The worst-case time- and space-complexity of the polyhedral computations we employ (e.g. convex hull and widening) is exponential in the number of dimensions (Khachiyan et al. 2006), i.e. the number of annotations in types. However, we remark that the number of annotations grows with the size of *types* but not the size of *programs*; admitting that the former remains bounded, we conjecture that the exponential complexity should not prevent the practical use of our analysis.

   In contrast, the worst-case complexity of Presburger constraint checking required by Hughes and Pareto is doubly-exponential on the number of variables; yet our previous assumption that types remain bounded means this is not, by itself, a prohibitive limitation. As far as we know, the stack and heap has not been implemented; however, the size-only system has been implemented with acceptable type checking times (Pareto 1998).

Finally, the amortised analysis of Hofmann and Jost requires solving a linear programming problem with theoretical and practical polynomial worst-case complexity.

8.2.2 Core Hume language and abstract machine

In Section 4.2 of Chapter 4 we presented a core subset of the Hume language with recursive first-order functions and algebraic data types. The main novelty in this language subset is a clear separation of values in the *expression* and *coordination* layers of the core language:

- values of expressions are "unlifted", that is, the void value "*" is allowed only in box outputs;
- expressions in box outputs must construct heap results that are disjoint from the input values.

These restrictions allow implementing communication wires using a simple region mechanism and use region-reseting to recycle heap space. Unlike previous implementations that use a copying collector (e.g. the Hume Abstract Machine (Hammond 2003)) this ensures that each state transition requires only a bounded amount of computation, making reasoning about costs at the language level simpler.

In Section 6.2 of Chapter 6 we presented an execution model for Core Hume in the form of an abstract machine using region memory management. Although it operates directly over expressions rather than compiled instructions, it accurately mimics stack and heap costs of a realistic implementation, e.g. by using a single stack for temporary values and call frames, by accounting stack costs for pattern matching and for tags required in heap values.

8.2.3 Extensions for optimisations

In Section 6.6 of Chapter 6 we have exposed some standard optimisations—namely, tail calls, unboxed representation of enumerations and explicit heap deallocation—as language-level annotations and extended both the abstract machine and static analysis to support them.

Dealing with optimisations in the context of resource-sensitive systems is important not only because it reduces absolute costs but also because it can improve predictability: our analysis was able to obtain bounds for optimised programs that exhibit linear costs when it could not do so for the naive supra-linear ones (e.g. the heap costs for the destructive list sorting algorithms of Section 7.2.5).

8.2.4 Experimental results

In Chapter 7 we conducted an experimental assessment of the cost bounds obtained by static analysis against profiling data collected from an implementation of the abstract machine. The examples chosen include some textbook functional algorithms (Section 7.2) and prototype embedded systems (Section 7.3).

Tables 7.1, 7.5, 7.6 and 7.7 summarise the quantitative assessment of the relative error of the predictions; the results indicate that our analysis yields *exact* bounds for simple examples and good upper-bounds for moderately complex ones (e.g. list sorting, red-black trees, lift controller and the geo-server examples).

8.2.5 Cost annotations and lifting

The initial naive formulation of our cost analysis in Section 6.4.3 introduced one disjunction for each syntax node in the program to account for the maximum stack costs of subexpressions; this results in generating an exponential number of constraints with respect to the program size. In order to obtain a more practical analysis we have extended the core language in Section 6.4.4 with *cost annotations* for assigning stack and heap costs to expressions; in retrospect, these are simply the “tick” annotations of a cost monad.

The purpose of cost annotations is to allow moving costs from sub-expressions to enclosing ones, synthesising constraints with maximum costs instead of disjunctions. This is accomplished by a *cost lifting* transformation defined in Section 6.4.6. In our prototype implementation of the analysis we left the decision of applying cost lifting to the user; this allowed us to conduct an experimental assessment of the technique. Furthermore, the cost lifting results in cost bounds that are no lower than the original, and therefore is always sound.

The usefulness of the transformation was verified experimentally with the red-black tree example of Section 7.2.7, Chapter 7; the results in Table 7.4 demonstrate a trade-off between the precision of cost bounds and the analysis time and space usage, e.g. the worst precision stack analysis has 5 times the relative error[2] of the

[2] But still yields a finite upper bound for stack.

best one but is 50 times faster and uses less than 1/10 of the memory. One advantage of this technique is that it allows losing precision in a gradual manner by specifying a cut-off depth for cost annotations; moreover, it can be applied to specific functions as required rather than the whole program.

8.2.6 Sized type analysis

We also identify two restricted contributions in the area of sized type analysis. The size type system of Chapter 5 is based on the one by Chin and Khoo (2001); however, the formulation of the size semantics and the formulation and soundness proofs of Theorems 5.21 and 5.23 are new:

1. we allow user-defined sizes for data types; the definition of the size function (Table 5.10 on page 122) and the soundness results (Theorems 5.21 and 5.23) have been extended with the assumptions Σ for the data constructors.
2. the soundness proof of Chin and Khoo assumes the existence of a minimal size constraint $\mathcal{S}$ for every value (including functions); however, this assumption does not hold for the lattice of constraints. Our soundness proof corrects this by defining the size $\mathcal{S}$ for non-functional values only and stating the size approximation for functions in a pointwise manner (Definition 5.10 on page 122).

8.3 Limitations

The principal limitations of our approach concern the range of language features covered and the quality of size and cost bounds:

- *We have consider only a first-order language.* This restriction was imposed by a technical requirement in the soundness proof of the size analysis of Chapter 5, namely, the inclusiveness of the type semantics (Lemma 5.16). It would, however, be possible to apply the cost analysis to some higher-order programs even without size information (see Section 8.4.1). Alternatively, defunctionalisation can be employed to transform the program into a first-order version (e.g. the geo-server example of Section 7.3.3).
- *Our analysis cannot infer information for elements inside collection types.* This limitation is caused by insufficient size polymorphism in the sized types of constructors (see Section 5.6.3). To obtain an automatic sized type reconstruction algorithm, we considered only let-bound polymorphism and have not addressed this limitation.

- *Our analysis can only synthesise size and cost bounds when these are expressible as linear inequations.* This limitation allows the use of computational solvers (e.g. the Parma Polyhedra Library) which support not just the automatic simplification of constraints, but also the algorithmic construction of invariants for recursive functions.

8.4 Further work

8.4.1 Higher order functions

We have restricted our size analysis to a first-order language as a consequence of our soundness proof: Lemma 5.16 (page 124) requires that the domain of values satisfies the ascending chain condition; this holds trivially for a flat domain of zero-order values but not for a function space. On the other hand, we have not found a counterexample of the lemma in the higher-order setting, so we do not know whether the result can be extended to a higher-order setting. We therefore leave the extension of size analysis for higher-order functions as an open problem.

We do, however, believe that it is possible to extend our analysis to predict costs of higher-order functions even without considering size information, i.e. as in the time system of Dornic et al. (1992).

Extending the core language and abstract machine

Extending our core language and abstract machine to support higher-order functions requires dynamic representations of functional values, i.e. *closures* (Landin 1964); these will typically be allocated in the heap and so have to be accounted in the cost analysis.

A closure for a function $\lambda x.\, e$ is a pair of the function's code and the *environment*, i.e. the values of free variables in $\lambda x.\, e$. The code is shared by all dynamic instances of the value and can be simply a pointer to a static area; the values of free variables are dynamic and hence account for the variable heap cost of the closure.

Environments are implemented in the original SECD machine as linked structures shared among different closures (Kogge 1991); this makes it difficult to account the relative contribution of each closure to the heap residency. Modern implementations of functional languages, use a flat representation of environments as contiguous blocks of values, e.g. the SML-New Jersey compiler (Appel 1992) and the STG-machine (Jones 1992)); the heap cost of a closure is then proportional the number of free variables in the lambda-abstraction, thus making it better suited

$$\frac{\begin{array}{c}\mathrm{FV}(\lambda x.\, e) = \{y_1,\, \dots,\, y_n\} \qquad u_i = E(y_i) \\ a \notin \mathrm{dom}(H) \qquad a \ \textbf{at}\ r \qquad u_i\ \textbf{at}\ r \\ E' = [y_1 \mapsto u_1,\, \dots,\, y_n \mapsto u_n]\end{array}}{\langle \mathsf{eval}(\lambda x.\, e) : C,\, E,\, S,\, H\rangle \overset{r}{\hookrightarrow} \langle C,\, E,\, a : S,\, H[a \mapsto \langle \lambda x.\, e, E'\rangle]\rangle} \tag{8.3}$$

$$\langle \mathsf{eval}(e_1\ e_2) : C,\, E,\, S,\, H\rangle \overset{r}{\hookrightarrow} \langle \mathsf{eval}(e_1) : \mathsf{eval}(e_2) : \mathsf{apply} : C,\, E,\, S,\, H\rangle \tag{8.4}$$

$$\frac{H(a) = \langle \lambda x.\, e, E'\rangle}{\begin{array}{l}\langle \mathsf{apply} : C,\, E, u : a : S,\, H\rangle \overset{r}{\hookrightarrow} \\ \qquad\qquad \langle \mathsf{bind}(x) : \mathsf{eval}(e) : \mathsf{ret}(1) : [],\, E',\, u : \langle C,\, E\rangle : S,\, H\rangle\end{array}} \tag{8.5}$$

Table 8.1: Core Hume machine extensions for higher-order functions.

for syntax-directed cost analysis. Moreover, experimental measurements show that shared representations of closures have no performance advantage and may lead to unexpected memory leaks (Appel 1992, chapters 12 and 15). We therefore choose a flat representation of closures for the Core Hume machine; this will be reflected in the heap cost metric for closures.

Consider the extension of the Core Hume language with expressions for functional abstraction and application:

$$e \quad ::= \quad \cdots \quad | \quad \lambda x.\, e \quad | \quad e_1\, e_2 \tag{8.1}$$

For simplicity, we consider only single argument applications; multiple arguments can be encoded in (heap) tuples. We also extend the boxed (heap) values of the Core Hume abstract machine with closures $\langle \lambda x.\, e, E\rangle$ where E is an environment, i.e. a mapping from variables to (unboxed) values.

$$b \quad ::= \quad \cdots \quad | \quad \langle \lambda x.\, e, E\rangle \tag{8.2}$$

Our abstract machine will maintain the invariant that closure environments are always *minimal*, i.e. for every closure $\langle \lambda x.\, e, E\rangle$ we have $\mathrm{dom}(E) = \mathrm{FV}(\lambda x.\, e)$; this means that the environment is implemented as a contiguous block of unboxed values. Letting $n = |\mathrm{FV}(\lambda x.\, e)|$, we we will therefore assume that a closure $\langle \lambda x.\, e, E\rangle$ has a heap cost of $2 + n$: one word for a tag, one word for a code pointer and n words for the values[3] of free variables.

Table 8.1 presents the extended machine transitions:

[3] These are unboxed integers or pointers to heap-allocated objects.

$$\frac{\Gamma, x : \tau' \vdash_{\mathsf{COST}} e : \tau \ \$ \ s \, ; h \mid \phi}{\Gamma \vdash_{\mathsf{COST}} \lambda x.\, e : \tau' \xrightarrow{s;h} \tau \ \$ \ s' \, ; h' \mid \phi \wedge s' = 1 \wedge h' = 2 + n} \quad n = |\mathrm{FV}(\lambda x.\, e)| \tag{8.6}$$

$$\frac{\Gamma \vdash_{\mathsf{COST}} e_1 : \tau' \xrightarrow{s_0;h_0} \tau \ \$ \ s_1 \, ; h_1 \mid \phi_1 \qquad \Gamma \vdash_{\mathsf{COST}} e_2 : \tau' \ \$ \ s_2 \, ; h_2 \mid \phi_2}{\begin{array}{r}\Gamma \vdash_{\mathsf{COST}} e_1 \ e_2 : \tau \ \$ \ s \, ; h \mid \phi_1 \wedge \phi_2 \wedge s = \max(s_0 + 2,\, s_1,\, s_2 + 1) \\ \wedge\, h = h_0 + h_1 + h_2\end{array}} \tag{8.7}$$

Table 8.2: Cost analysis rules for higher-order abstraction and application.

- Rule (8.3) specifies the evaluation of a lambda-abstraction: a new closure is allocated and the address pushed on top of the stack. Note that we impose side conditions to prevent cross-region references in the closure environment; this is done to avoid possibility of dangling references after region-resetting (see Section 6.2).

- Rule (8.4) specifies the evaluation applications: first evaluate the function obtaining a closure, then the argument and finally evaluate the closure; we introduce a new apply machine directive for the final step.

- Rule (8.5) implements the apply transition: fetch the closure and argument from the stack, push a continuation, evaluate the function body under an augmented environment and restore the continuation.

Extending the cost analysis

Table 8.2 presents extended type and effect judgements with costs of closure creation and application. Rule (8.6) specifies the cost of creating a new closure for an abstraction with n free variables, namely, $2 + n$ heap cells and one stack cell. The stack and heap costs of the body are simply transposed into latent costs. Rule (8.7) expresses the costs for an application as a combination of costs for the function and argument parts plus the latent costs. The constant offsets of stack costs $s_0 + 2$ and $s_2 + 1$ reflect the evaluation order chosen in Table 8.1: the argument is evaluated on a stack extended by one word (for the closure); the latter is evaluated on a stack extended by two words (the argument and continuation).

While we have not formally proved the correctness of the rules in Table 8.2, we believe they should follow the same methodology used for proving Lemma 6.4 of Chapter 6.

This extension should be sufficiently expressive for obtaining costs of simple higher-order combinators. For example, for the higher-order composition function

$$\mathsf{let}\ compose\ (f,\, g,\, x) = f\ (g\ x)\ \mathsf{in}\ \ldots$$

we would derive a type annotated with costs but no sizes:

$$compose : \forall s_1, h_1, s_2, h_2, s, h.\ \langle \forall abc.\ (b \xrightarrow{s_1;h_1} c) \times (a \xrightarrow{s_2;h_2} b) \times a \xrightarrow{s;h} c,$$
$$s = \max(s_1, s_2) + \cdots \wedge h = h_1 + h_2 + \cdots \rangle$$

where a, b, c are *unsized* type variables.

This extension should also be sufficient for obtaining costs of a higher-order version of the geo-server example of Section 7.3.3, i.e. without requiring the defunctionalisation of regions. It would not, however, be sufficient for obtaining costs of functions that traverse recursive data structures, e.g. the list *map* function with the following annotated type:

$$map : (a \xrightarrow{s';h'} b) \times \mathsf{List}^n\, a \xrightarrow{s;h} \mathsf{List}^m\, b$$

The difficulty is that the heap cost h of map is the *product* of two variables (the latent cost h' and list length n) and thus not expressible as a linear constraint.

8.4.2 Time analysis

Our framework of cost annotations should allow extending our analysis to account for time. The treatment of time should be similar to that of heap except for the absence of deallocation. To allow flexibility in cost assignment, we would treat time costs via annotations as we did for stack and heap: $\mathsf{time}^k\ \widehat{e}$ assigns a time cost k (expressed in suitable units) to expression e. To extend our analysis to predict *real-time* bounds we propose adapting the methodology of Bonenfant et al. (2007):

1. design a virtual instruction set and associated compiler for our Core Hume abstract machine; in particular, variables need to be replaced by relative stack offsets (e.g. de Brujin indices); this is a straightforward exercise in language implementation that consists essentially of a sequence of program transformations (Ager et al. 2003*a*).

2. implement the virtual machine interpreter in the concrete machine, e.g. directly or using C compiler;

3. use the *aiT* analysis tool (Ferdinand et al. 1999, 2001) to obtain guaranteed worst-case time of each virtual instruction for the target architecture, e.g. in processor cycles;

4. transpose the time costs obtained for virtual instructions to the source-level analysis using time cost annotations; this assignment can be done trivially by modifying the compiler to annotate expressions instead of generating virtual code.

Worst-case time bounds for individual virtual instructions are likely to be too pessimistic because little is known in advance about the cache and pipeline states. To improve precision, it is possible to consider the basic blocks of virtual instructions generated for specific programs.

We believe this approach should, in principle, yield at least as accurate time costs as the ones obtained for the HAM machine by Bonenfant et al. (2007) because each transition in our abstract machine performs only a bounded amount of computation. Hence, each virtual instruction should compile into machine code performing sequencial or (at most) statically-bounded loops computations, which fit the capabilities of the *aiT* tool.

However, one important limitation still applies: the cost analysis is only able to synthesise linear inequations, so we can only obtain upper bounds for linear-time algorithms.

8.4.3 Polynomial cost bounds

An interesting research direction is to explore recent proposals for abstract interpretation with polynomial inequalities of bounded degree (Bagnara et al. 2005). This technique approximates non-linear terms in polynomials as additional independent variables and uses convex polyhedra to represent this extended algebraic structure.

Unlike methods based on quantifier elimination, which are computationally-prohibitive, e.g. Kapur (2004), the proposal of Bagnara et al. can be implemented using the same underlying machinery of convex polyhedra used for linear relation analysis. Moreover, it can express not just polynomial equalities but also *inequalities* which are essential for cost approximations.

The experimental results of Bagnara et al. show that the technique can obtain quadratic polynomial invariants in reasonable time; we conjecture that this technique can be used to extend our analysis to obtain low-degree (e.g. quadratic or cubic) polynomial cost bounds.

Appendix A

Mathematical notation

In this appendix we review some mathematical notation and results that are used in this thesis. Since all the notions and results are well-known, we include them only to fix notation and refer the reader to standard textbooks on mathematical foundations of programming language semantics for a thorough presentation (Mitchell 1996, Winskel 1993, Davey and Priestley 1990).

A.1 Partially ordered sets

Partial orders play a pivotal role in both denotational semantics of programming languages (Stoy 1977, Winskel 1993) and program analyses (Nielson et al. 1999). We briefly review some definitions and elementary results that are used in this thesis.

A.1.1 Order relations

A *pre-order* on a set A is a binary relation $\sqsubset \subseteq A \times A$ that is reflexive and transitive. A *partial order* on A is a pre-order that is also *antisymmetric*, i.e. a binary relation $\sqsubseteq \subseteq A \times A$ that is reflexive, transitive and antisymmetric.

A.1.2 Bottom and top elements

Let $(A, \sqsubseteq)$ be a partial order; if A has a smallest element, i.e. an element $\perp \in A$ such that $\perp \sqsubseteq x$ for all $x \in A$, then we say that $\perp$ is the *bottom element* of A. Dually, if A has a greatest element, it is designated the *top element* and represented by $\top$. It follows from the antisymmetry of $\sqsubseteq$ that the bottom and top elements, if they exist, are unique.

A.1.3 Least upper-bounds and greatest lower-bounds

Let $(A, \sqsubseteq)$ be a partial order and $X \subseteq A$; $u \in A$ is *upper-bound* of X iff $x \in X$ implies $x \sqsubseteq u$. Dually, $l \in A$ is a *lower-bound* of X iff $x \in X$ implies $l \sqsubseteq x$.

The *least upper-bound* (or *join*) of X is $u \in A$ such that (1) u is an upper-bound of X; (2) for all u' such that u' is an upper-bound of X, we have $u \sqsubseteq u'$. We write $\bigsqcup X$ for the least upper-bound of X whenever it exists (in which case it is unique) and $\bigsqcap X$ for the dual notion of *greatest lower-bound* (also called *meet*).

A.1.4 Lattices

A *lattice* is a partial order $(D, \sqsubseteq)$ such that every pair of elements $x, y \in D$ has a least upper bound $x \sqcup y \in D$ and greatest lower bound $x \sqcap y \in D$.

A *complete lattice* is a partial order $(D, \sqsubseteq)$ such that every subset $X \subseteq D$ has a least upper bound $\bigsqcup X \in D$ and greatest lower bound $\bigsqcap X \in D$. It is straightforward to check that every complete lattice must have a least element $\bot = \bigsqcap D = \bigsqcup \emptyset$ and a greatest element $\top = \bigsqcup D = \bigsqcap \emptyset$.

A.1.5 Complete partial orders

A countable subset $X = \{x_0, x_1, x_2, \ldots\}$ of a partial order $(D, \sqsubseteq)$ is an *ω-ascending chain* (or simply *ascending chain*) iff $X \subseteq D$ and $x_i \sqsubseteq x_{i+1}$ for all $i \in \mathbb{N}$.

A partial order $(D, \sqsubseteq)$ is *complete* (a *CPO*) if and only if every ω-ascending chain in D has a least upper-bound in D, i.e. $x_i \in D$ and $x_i \sqsubseteq x_{i+1}$ for all $i \in \mathbb{N}$ implies $\bigsqcup\{x_i : i \in \mathbb{N}\} \in D$.

Our definition of CPO follows Winskel (1993) and Mitchell (1996) and does not require the existence of least elements, and thus corresponds to the notion of *pre-CPO* in Stoy (1977) and Davey and Priestley (1990). We designate by *pointed CPO* a CPO with least element.

It is immediate to verify that every complete lattice is a (pointed) CPO; the converse implication does not hold because a CPO need not have a top element.

A.1.6 Continuity and fixed points

Let $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ be CPOs. A function $f : A \to B$ is *monotone* if and only if $x \sqsubseteq_A y$ implies $f(x) \sqsubseteq_B f(y)$ for all x, y. A function f is *continuous* if and only if f is monotone and $f(\bigsqcup_A X) = \bigsqcup_B f(X)$ for all ascending chains $X \subseteq A$, i.e. f preserves least-upper bounds of chains.

If $f : A \to A$ is a continuous function on a pointed CPO $(A, \sqsubseteq)$ then it has a unique *least fixed-point*, which we designate by $\mathit{fix}(f)$.

Theorem A.1 (Fixed point theorem) *Let $(A, \sqsubseteq)$ be a pointed CPO and $f : A \to A$ a continuous function. Define*

$$\mathit{fix}(f) = \bigsqcup_{n \in \mathbb{N}} f^n(\bot) .$$

Then $\mathit{fix}(f)$ is the least-fixed point of f i.e. $f(\mathit{fix}(f)) = \mathit{fix}(f)$ and if $f(x) \sqsubseteq x$ then $\mathit{fix}(f) \sqsubseteq x$.

Proof: See, for example, Davey and Priestley (1990, chapter 4, page 89). □

A.1.7 Constructing complete partial orders

We now review some standard constructions of CPOs.

Discrete order: Any set A is a CPO under the *discrete partial order* $x \sqsubseteq y \overset{\text{def}}{\iff} x = y$.

Lifting: We can turn any CPO $(A, \sqsubseteq)$ into a pointed CPO $(A_\bot, \sqsubseteq')$ by adding a distinct element $A_\bot = A \cup \{\bot\}$ with $\bot \notin A$ and defining $x \sqsubseteq' y \overset{\text{def}}{\iff} x = \bot \vee x \sqsubseteq y$.

Cartesian product: If $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ are CPOs, then $(A \times B, \sqsubseteq)$ is a CPO with the ordering $(x, y) \sqsubseteq (x', y') \overset{\text{def}}{\iff} x \sqsubseteq_A x' \wedge y \sqsubseteq_B y'$. $(A \times B, \sqsubseteq)$ is a pointed CPO whenever $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ are pointed: if $\bot_A$, $\bot_B$ are, respectively, the least elements of A and B, then the least element of $A \times B$ is $(\bot_A, \bot_B)$.

Continuous function space: If $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ are CPOs, then $[A \to B]$ is the set of continuous functions from A to B. If B is pointed then $([A \to B], \sqsubseteq)$ is a pointed CPO under the *pointwise ordering*:

$$f \sqsubseteq g \overset{\text{def}}{\iff} \forall x \in A\ f(x) \sqsubseteq_B g(x)$$

The bottom element of $[A \to B]$ is the constant function that yields $\bot_B$, which we represent by $\bot_{[A \to B]}$ or simply $\bot$ when the carrier set $[A \to B]$ is clear from the context.

A.1.8 Ascending chain condition

A partial order $(D, \sqsubseteq)$ satisfies the *ascending chain condition* (ACC) if and only if every ascending chain in D eventually stabilises, i.e. for all chains $\{(x_i)_{i \in \mathbb{N}}\}$ in D, we have:

$$\exists k \in \mathbb{N}\ \forall i \in \mathbb{N}\ (i \geq k \implies x_i = x_k)$$

Any discrete partial order satisfies (ACC); and if $(A, \sqsubseteq_A)$, $(B, \sqsubseteq_B)$ both satisfy (ACC) then so do $A_\perp$, $B_\perp$ and $A \times B$. However, $[A \to B]$ might *not* satisfy (ACC) even if A and B do. For a counter-example, let $A = \mathbb{N}$ with the discrete order and $B = \{\perp, \top\}$ the two-point lattice; then

$$f_i(j) \overset{\text{def}}{=} \begin{cases} \top, & \text{if } i < j \\ \perp, & \text{otherwise} \end{cases}$$

defines an ascending chain of continuous functions $\{f_i : i \in \mathbb{N}\}$ with least upper bound the constant function that yields $\top$. But $f_i(i) = \perp$ and $f_{i+1}(i) = \top$, therefore $f_i \neq f_{i+1}$ for all i, which shows that the chain is not stabilising.

A.1.9 Ideals

Let $(D, \sqsubseteq)$ be a CPO; a subset $X \subseteq D$ is *downwards closed* iff whenever $x \in X$ and $y \sqsubseteq x$ then $y \in X$. A subset $X \subseteq D$ is *consistently closed* iff whenever $Y \subseteq X$ and $\bigsqcup Y$ exists in D, then $\bigsqcup Y \in X$.

A non-empty subset X of a CPO $(D, \sqsubseteq)$ is an *ideal* iff X is downwards closed and consistently closed. The set of all ideals of D is designated $\mathcal{I}(D)$.

Lemma A.2 *If $(D, \sqsubseteq)$ is a pointed CPO then $(\mathcal{I}(D), \subseteq)$ is a pointed CPO.*

It is straightforward to verify that the singleton $\{\perp_D\}$ and D are, respectively, is the bottom and top elements of $(\mathcal{I}(D), \subseteq)$.

A.2 Interval constraints and solutions

In this section we formally define the interval constraints used in the coordination layer analysis of Section 6.7, prove the existence of minimal solutions and describe an algorithm for obtaining a solution.

A.2.1 Interval constraints

An *interval constraint* $\ell \sqsupseteq \phi$ is pair of a variable ℓ and a convex size constraint ϕ (see Section 5.5). A *system* of interval constraints is a finite set $\{(\ell_i \sqsupseteq \phi_i)_{i=1}^N\}$ where the variables ℓ_i on the left-hand sides are not necessarily distinct.

Informally, a constraint $\ell \sqsupseteq \phi$ expresses a lower-bound for ℓ as the projection of a formular ϕ over variable ℓ; to define this formally, we begin by defining the solutions of interval constraints.

A *solution* of a system of interval constraints is an assignment $\mathcal{V} : \mathbf{ZVar} \to \mathbf{Interval}$ of intervals to variables. The *projection* of a constraint ϕ over a variable ℓ under a solution $\mathcal{V}$ is an element of **Interval** defined by:

$$[\![\phi]\!]_\ell\, \mathcal{V} \stackrel{\text{def}}{=} \Big\{ z \in \mathbb{Z} \,:\, [\ell \mapsto z] \models \exists \vec{\ell'}.\, \Big(\phi \wedge \textstyle\bigwedge_{1\le i\le n} \inf \mathcal{V}(\ell'_i) \le \ell'_i \le \sup \mathcal{V}(\ell'_i)\Big) \Big\} \quad \text{where } \{\ell'_1, \ldots, \ell'_n\} = \mathrm{FZV}(\phi) \setminus \{\ell\} \tag{A.1}$$

In (A.1) we consider $\inf \bot = +\infty$ and $\sup \bot = -\infty$; also, inequalities of the form $\ell \le \infty$ and $-\infty \le \ell$ are interpreted as true, while inequalities such as $\ell \le -\infty$ and $\infty \le \ell$ are interpreted as false; these conventions deal with empty intervals with no need for special cases.

We remark that projection is well-defined because the expression on the right-hand side of $\models$ in (A.1) yeilds the same set regardless of the choice of ordering among the free variables of ϕ. Second, note that when ϕ is a convex constraint (i.e. a conjunction of linear inequalities) then the projection defines an convex subset of $\mathbb{Z}$, i.e. an interval. We abuse notation and take the result of (A.1) as an element of **Interval**.

A solution $\mathcal{V}$ *satisfies* a constraint $\ell \sqsupseteq \phi$ if and only if the interval $\mathcal{V}(\ell)$ contains the projection $[\![\phi]\!]_\ell\, \mathcal{V}$. Formally, we write $\mathcal{V} \models \ell \sqsupseteq \phi$ and define it by[1]:

$$\mathcal{V} \models \ell \sqsupseteq \phi \stackrel{\text{def}}{\iff} \mathcal{V}(\ell) \sqsupseteq [\![\phi]\!]_\ell\, \mathcal{V} \tag{A.2}$$

Finally, a solution satisfies a system if it satisfies every constraint:

$$\mathcal{V} \models \{(\ell_i \sqsupseteq \phi_i)_{i=1}^N\} \stackrel{\text{def}}{\iff} \textstyle\bigwedge_{1\le i\le N} \mathcal{V} \models \ell_i \sqsupseteq \phi_i \tag{A.3}$$

A.2.2 Existence of solutions

A system of interval constraints $\{(\ell_i \sqsupseteq \phi_i)_{i=1}^N\}$ always has a trival solution, namely the assignment $\mathcal{V}(\ell) \stackrel{\text{def}}{=} [-\infty, +\infty]$ for all ℓ. This is the *greatest* solution with respect to $\sqsubseteq$ and corresponds to the least informative analysis. Dually, the most informative analysis corresponds to the *least* solution with respect to $\sqsubseteq$. We will show that a unique least solution exists by showing the "model intersection property" for satisfiability. First, we prove an auxiliary result concerning projection.

[1] We abuse the symbol $\sqsupseteq$ and use it on the left of $\Leftrightarrow$ as the *syntatic* connective in a constraint and on the right as the *semantic* relation of interval inclusion.

Lemma A.3 $[\![\phi]\!]_\ell\,(\bigsqcap_i \mathcal{V}_i) = \bigsqcap_i([\![\phi]\!]_\ell\,\mathcal{V}_i)$ *i.e. projection is a complete meet-morphism.*

Proof: We need to prove the equality of two sets defined by predicates that differ only in the bounds of the free variables of ϕ. Consider a variable $\ell' \in \mathrm{FZV}(\phi)$, $\ell' \neq \ell$; the bounds for ℓ' in the definition of $[\![\phi]\!]_\ell\,(\bigsqcap_i \mathcal{V}_i)$ are

$$\inf\,(\textstyle\bigsqcap_i \mathcal{V}_i)(\ell') \leq \ell' \leq \sup\,(\textstyle\bigsqcap_i \mathcal{V}_i)(\ell')$$

But

$$\begin{aligned} \inf\,(\textstyle\bigsqcap_i \mathcal{V}_i)(\ell') &= \max\,\{\inf\, \mathcal{V}_i(\ell') \ :\ i \in \mathbb{N}\} \\ \sup\,(\textstyle\bigsqcap_i \mathcal{V}_i)(\ell') &= \min\,\{\sup\, \mathcal{V}_i(\ell') \ :\ i \in \mathbb{N}\} \end{aligned}$$

Substitution in the previous equation yields:

$$\begin{aligned} &\max\,\{\inf\, \mathcal{V}_i(\ell') : i \in \mathbb{N}\} \leq \ell' \leq \min\,\{\sup\, \mathcal{V}_i(\ell')) : i \in \mathbb{N}\} \iff \\ \iff &\forall i \in \mathbb{N}. \quad \inf\, \mathcal{V}_i(\ell') \leq \ell' \leq \sup\, \mathcal{V}_i(\ell') \end{aligned}$$

The last equation caracterizes the bounds of ℓ' in the definition of $\bigsqcap_i([\![\phi]\!]_\ell\,\mathcal{V}_i)$ as required. □

Corollary A.4 $\models$ *has the model intersection property, i.e.*

$$(\forall i \in \mathbb{N}.\ \mathcal{V}_i \models \ell \sqsupseteq \phi) \implies \textstyle\bigsqcap_i \mathcal{V}_i \models \ell \sqsupseteq \phi$$

Proof: The hypothesis is $\mathcal{V}_i(\ell) \sqsupseteq [\![\phi]\!]_\ell\,\mathcal{V}_i$ for all $i \in \mathbb{N}$. Since **Interval** is a complete lattice, this implies $\bigsqcap_i \mathcal{V}_i(\ell) \sqsupseteq \bigsqcap_i([\![\phi]\!]_\ell\,\mathcal{V}_i)$. By Lemma A.3, this is equivalent to $\bigsqcap_i \mathcal{V}_i(\ell) \sqsupseteq [\![\phi]\!]_\ell\,(\bigsqcap_i \mathcal{V}_i)$, which is the desired result. □

The model intersection property implies that constraint systems have a least solution, namely the meet of the solutions of all constraints: such a set is countable and not empty (it contains at least the trivial solution); therefore, by Corollary A.4, the meet is also a solution. From the properties of the lattice, it is immediate that this solution is unique and minimal.

A.2.3 Iterative approximation of solutions

The projection function is monotone over solutions, that is, we cannot invalidate a constraint by moving to a larger solution.

$$\forall\phi.\ \forall\ell.\ \forall\mathcal{V}.\ \forall\mathcal{V}'. \quad \mathcal{V} \sqsubseteq \mathcal{V}' \implies [\![\phi]\!]_\ell\,\mathcal{V} \sqsubseteq [\![\phi]\!]_\ell\,\mathcal{V}' \tag{A.4}$$

Monotonicity suggests that might employ a method based on *chaotic iteration* for solving a constraint system; these are typically used for solving constraint systems resulting from program analyses (Nielson et al. 1999, chapter 6).

1. start with the least solution $\mathcal{V}(\ell) := \bot$ for all variables ℓ;
2. while there exists i such that $\mathcal{V}(\ell_i) \not\sqsupseteq [\![\phi_i]\!]_{\ell_i}\, \mathcal{V}$:
   assign $\mathcal{V}(\ell_i) := \mathcal{V}(\ell_i) \sqcup [\![\phi_i]\!]_{\ell_i}\, \mathcal{V}$.
3. terminate with answer $\mathcal{V}$

If the iteration of step 2 terminates, then $\mathcal{V}$ satisfies $\{(\ell_i \sqsupseteq \phi_i)_{i=1}^{N}\}$. Since the lattice of intervals is complete, the solution obtained is unique and in fact the least solution with respect to interval ordering.

However, there is no guarantee that the iteration in the lattice of intervals will terminate because the former does not respect the ascending chain condition. In this case the solution to employ a widening operator to accelerate converge to a post-fixed point (see Section 2.3.3). Let ∇ be a widening operator for intervals (e.g. the one defined in Section 2.3.5); to perform the iteration with widening we simply replace the upper-bound $\sqcup$ in step 2 by ∇:

2. while there exists i such that $\mathcal{V}(\ell_i) \not\sqsupseteq [\![\phi_i]\!]_{\ell_i}\, \mathcal{V}$:
   assign $\mathcal{V}(\ell_i) := \mathcal{V}(\ell_i) \nabla [\![\phi_i]\!]_{\ell_i}\, \mathcal{V}$.

Example A.5 Consider the cyclic system of constraints $\{i \sqsupseteq i = 0, i \sqsupseteq i = j{+}1, j \sqsupseteq j = i\}$ envolving two variables i, j. The method chaotic iteration constructs the following ascending sequence increasing intervals for i and j:

$$\bot \sqsubseteq [0,\, 0] \sqsubseteq [0,\, 1] \sqsubseteq [0,\, 2] \sqsubseteq \ldots$$

This iteration does not converge in finite number of steps to the least solution $[0, +\infty]$. By contrast, the iteration with the widening operator of Section 2.3.5 yields

$$\bot \sqsubseteq [0,\, 0] \sqsubseteq [0,\, 1] \sqsubseteq [0,\, +\infty]$$

and stabilises after the second iterate. □

A.2.4 Practical implementation

A number of improvements can be made to the method outlined in the previous section to obtain a practical and efficient implementation. Traversing the constraints according to dependencies can reduce the amount of work to be done; moreover, iterative approximations are required only for the truly cyclic components; therefore, the natural choice is to start by partitioning constraints into strongly connected components and iterate over those (Nielson et al. 1999, chapter 6, pages 381–384).

A second possibility is to employ a more refined widening operator for intervals: the naive widening introduced in Section 2.3.5 looses all precision of interval bounds in a single step. A more refined variant is obtained by considering a gradual widening of bounds given by a set K of integer constants, e.g. a set of integers explicitly mentioned in the program text or as user annotations. Define the operator ∇_K parametrised by K as

$$\begin{aligned}
\bot \nabla_K \mathit{int} &= \mathit{int} \nabla_K \bot = \mathit{int} \\
[z_1,\, z_2] \nabla_K [z_1',\, z_2'] &= [l,\, u] \\
\text{where } l &= \begin{cases} z_1 & \text{if } z_1 \le z_1' \\ \max\{k \in K \cup \{-\infty\} \,:\, k \le z_1'\} & \text{otherwise} \end{cases} \\
u &= \begin{cases} z_2 & \text{if } z_2' \le z_2 \\ \min\{k \in K \cup \{+\infty\} \,:\, z_2' \le k\} & \text{otherwise} \end{cases}
\end{aligned}$$

Then ∇_K is a also widening operator (Nielson et al. 1999, pages 228–229); in fact, the simpler widening is a special case of the above definition when $K = \emptyset$.

We have implemented a solver with both these techniques (iterating through strongly-connected components and widening with constants) as part of our size and cost analysis, and have found it to yield quite accurate results for the coordination layer analysis of the embedded system examples in Section 7.3.

Appendix B

Program listings

This appendix contains the listings for the Core Hume examples, namely the mine pump controller, the lift controller and the geometric region server.

B.1 Mine pump controller

```
{-
  Simplified mine pump controller

  This is a simplified controller for a pump operating
  in a mine subject to dangerous gases like methane (CH4)
   and carbon monoxide (CO).

  The pump can operate in two modes: auto or manual;
  default is auto.  In auto mode, the pump is operated
  when the water level goes high  (indicated by high_water_level)
  and is switched off when the level goes low. In manual mode,
  the pump is turned on or off under the control of the operator.

  At no time should the pump be running when the methane level
  reaches a critical value. When the methane level or the
  carbone monoxide level reached critical values an appropriate
  alarm is raised; when the methane level reaches its critical
  value, the pump is turned off.

  Reference: A. Burns, A. Wellings, Real-Time Systems and
  their Programming Languages, Addison Wesley, 1990
-}


-- pump operation mode
data MODE = AUTO | MANUAL ;
```

```
-- control signal to the pump engine to turn on and off
data STATE = ON | OFF ;

-- copy the mode and control data types
copyMODE :: MODE -> MODE;
copyMODE AUTO = AUTO;
copyMODE MANUAL = MANUAL ;

copySTATE :: STATE -> STATE;
copySTATE ON = ON;
copySTATE OFF = OFF;

-- operator requests
data REQ = REQ_ON | REQ_OFF | REQ_AUTO | REQ_MANUAL ;

-- low and high water level sensors
type WATER = (Bool,Bool) ;

--
-- the pump controller box
--
box pump
in (mode::MODE, state::STATE, water_sensors::WATER,
     ch4_alarm::Bool, req::REQ)
out (mode'::MODE, state'::STATE, log::String)
unfair
  (mode, ON, water, True, *) ->
       (copyMODE mode, OFF, "CH4 alarm: turning pump off")
| (AUTO, state, water, alarm, REQ_MANUAL) ->
       (MANUAL, copySTATE state, "Operator requested MANUAL mode")
| (AUTO, state, water, False, _*) ->
       (AUTO, auto_control state water, "Running in auto mode")
| (MANUAL, state, water, False, REQ_ON) ->
       (MANUAL, ON, "Operator requested pump ON")
| (MANUAL, state, water, alarm, REQ_OFF) ->
       (MANUAL, OFF, "Operator requested pump OFF")
| (MANUAL, state, water, alarm, REQ_AUTO) ->
       (AUTO, copySTATE state, "Operator requested AUTO mode")
-- default rule
| (mode, state, water, alarm, _*) -> (copyMODE mode, copySTATE state, *)
;

wire pump.mode pump.mode';
wire pump.state pump.state';

initial pump.state OFF;
initial pump.mode AUTO;
initial pump.water_sensors (False,False);
```

```
-- automatic pump control based on readings from water level sensors
auto_control :: {STATE,WATER} -> STATE;
auto_control OFF (_,high)  = if high then ON else OFF;
auto_control ON  (low,_)   = if low then OFF else ON;


-------------------------------------------------------------------------
-- monitor the environment
-- triggers alarms in appropriate conditions
-------------------------------------------------------------------------
box environment
in (ch4_level::Float, co_level::Float, airflow_level::Float)
out (ch4_alarm::Bool, ch4_alarm'::Bool,
     co_alarm::Bool, airflow_alarm::Bool)
match
  (ch4, co, air) -> (ch4>.high_ch4_level, ch4>.high_ch4_level,
                     co>.high_co_level, air<.low_airflow_level)
;

high_ch4_level :: Float;
high_ch4_level = 15.0;

high_co_level :: Float;
high_co_level = 15.0 ;

low_airflow_level :: Float;
low_airflow_level = 4.0;


wire environment.ch4_alarm pump.ch4_alarm;
wire environment.ch4_alarm' logger.ch4_alarm;
wire environment.co_alarm logger.co_alarm;
wire environment.airflow_alarm logger.airflow_alarm;


------------------------------------------------------------------------------
-- simulators for environment sensors
------------------------------------------------------------------------------
box water
in (time::Float)
out (time'::Float, sensors::WATER)    -- (lo,hi) booleans
match
   (t) -> (t+.0.2,
           let wl = 10.0+.10.0*.(sin# t) in
           (wl<.low_water_level, wl>.high_water_level))
;


low_water_level :: Float;
low_water_level = 2.0;

high_water_level :: Float;
high_water_level = 10.0;


box methane
in (time::Float)
out (time'::Float, level::Float)
match
    (t) -> (t+.0.1, 8.0+.10.0*.(sin# (0.5+.t)))
;

box carbmonoxide
in (time::Float)
out (time'::Float, level::Float)
match
   (t) -> (t+.0.2, 8.0+.10.0*.(sin# (0.3 +. t)))
;

box airflow
in (time::Float)
out (time'::Float, level::Float)
match
  (t) -> (t+.0.3,  10.0+.10.0*.(sin# (0.6 +. t)))
;

wire water.time water.time';
wire methane.time methane.time';
wire carbmonoxide.time carbmonoxide.time';
wire airflow.time airflow.time';

wire water.sensors pump.water_sensors;
wire methane.level environment.ch4_level;
wire carbmonoxide.level environment.co_level;
wire airflow.level environment.airflow_level;

initial water.time 0.0;
initial methane.time 0.0;
initial carbmonoxide.time 0.0;
initial airflow.time 0.0;

-------------------------------------------------------------------------
-- simulator for operator requests
-------------------------------------------------------------------------
box operator
in (time::Int)
out (time'::Int, trigger::Bool)
match
  (t) -> (t+1, (rem# t 50) == 0)
```

```
;

box request
in (trigger::Bool)
out (req::REQ)
match
   (True) -> (let k = rem# (abs (randomInt# 0)) 4
              in  mkREQ k)
|  (False) -> (*)
;

abs :: Int -> Int;
abs x = if x>=0 then x else (negate# x);

mkREQ :: Int -> REQ;
mkREQ n = if n==0 then REQ_OFF
          else if n==1 then REQ_ON
          else if n==2 then REQ_AUTO
          else REQ_MANUAL;

wire operator.time operator.time';
wire operator.trigger request.trigger;
wire request.req pump.req;

initial operator.time 0;

---------------------------------------------------------------------------
-- logger box
---------------------------------------------------------------------------
box logger
in (ch4_alarm::Bool, co_alarm::Bool, airflow_alarm::Bool, pump::String)
out (dummy::Int)
fair
  (True, *, *, *) -> (putStrLn# "ALARM: CH4 level too high")
| (False, *, *, *)-> (*)
| (*, True, *, *) -> (putStrLn# "ALARM: CO level too high")
| (*, False, *, *)-> (*)
| (*, *, True, *) -> (putStrLn# "ALARM: airflow level too low")
| (*, *, False, *)-> (*)
| (*, *, *, str)  -> (putStrLn# str)
;

box sink
in (dummy::Int)
out (dummy':: Int)
match
  (_*) -> (*)
;

wire logger.dummy sink.dummy;
wire pump.log logger.pump;
```

B.2 Lifts controller

```
{- Two lift simulator based on an example by Greg Michaelson
   Pedro Vasconcelos 2006, 2007
 -}

type Floor = Int;       -- floor number, starting at 0

type Pending = [Bool];  -- list of floors to be visited

-- lift movement: going up, down, waiting or stopped
data Move = UP | DOWN | WAIT_UP | WAIT_DOWN | STOP ;

-- lift state
-- movement, current floor, lo/hi floors, pending requests
type State = (Move, Int, Int, Int, Pending) ;

-- request either lift
data Request = LIFT1 Int | LIFT2 Int;

-- number of floors
floors :: Int;
floors = 10;

-- create a list of n items
repeat :: {Int,a} -> [a];
repeat n x = if n==0 then [] else x:repeat (n-1) x ;

-- copy a state (not shared with the input)
copy_State :: State -> State;
copy_State (state, floor,  lo, hi, pending)
    = (copy_Move state, floor, lo, hi, copy_Pending pending);

copy_Move :: Move -> Move;
copy_Move UP   = UP;
copy_Move DOWN = DOWN;
copy_Move WAIT_UP = WAIT_UP;
copy_Move WAIT_DOWN = WAIT_DOWN;
copy_Move STOP = STOP;

copy_Pending :: Pending -> Pending;
copy_Pending [] = [];
copy_Pending (p:ps) = copy_Bool p : copy_Pending ps;

copy_Bool :: Bool -> Bool;
copy_Bool True = True;
copy_Bool False = False;

-- initial configuration (with n floors)
initialState :: Int -> State;
initialState n = (STOP, 0,0,0, repeat n False);

-- test if some boolean value is true
any :: Pending -> Bool;
any []         = False;
any (True:xs)  = True;
any (False:xs) = any xs;

abs :: Int -> Int;
abs x = if x>=0 then x else 0-x;

max :: {Int,Int} -> Int;
max x y = if x>=y then x else y;

min :: {Int,Int} -> Int;
min x y = if x>=y then y else x;

-- get the nth element of a list
index :: {Int,[a]} -> a;
index n (x:xs) = if n==0 then x else index (n-1) xs;

-- modify the nth element of a list
modify :: {Int,a,[a]}->[a];
modify n y (x:xs) = if n==0 then (y:xs) else x:modify (n-1) y xs;

set :: {Floor,Pending} -> Pending;
set n p =  modify n True p;

reset :: {Floor,Pending} -> Pending;
reset n p =  modify n False p;

-- request a lift to visit a floor
pragma "nohull"
req_lift :: {State,Floor} -> State;
req_lift (STOP, floor, lo,hi, pending) req
       = if req>floor then
             (WAIT_UP, floor, req,req, set req pending)
         else if req<floor then
             (WAIT_DOWN, floor, req,req, set req pending)
         else (STOP, floor, floor,floor, pending);
req_lift (state, floor, lo,hi, pending) req
       = (state, floor, min lo req, max hi req, set req pending);


-- move a lift
pragma "nohull"
```

```
move_lift :: State  -> State;
move_lift (UP, floor, lo,hi, pending)
       = if index floor pending then
            (WAIT_UP, floor, lo,hi, reset floor pending)
         else if floor<hi then
            (UP, floor+1, lo,hi, pending)
         else
            (WAIT_DOWN, floor, lo,hi, pending);
move_lift (DOWN, floor, lo,hi, pending)
       = if index floor pending then
            (WAIT_DOWN, floor, lo,hi, reset floor pending)
         else if floor>lo then
            (DOWN, floor-1, lo,hi, pending)
         else
            (WAIT_UP, floor, lo,hi, pending);
move_lift (WAIT_UP, floor, lo,hi, pending)
       = if any pending then
            (UP, floor, lo,hi, pending)
         else
            (STOP, floor, floor, floor, pending);
move_lift (WAIT_DOWN, floor, lo,hi, pending)
       = if any pending then
            (DOWN, floor, lo,hi, pending)
         else
            (STOP, floor, floor, floor, pending);
move_lift (STOP, floor, lo, hi, pending)
       = (STOP, floor, lo, hi, pending);

-- distance between a lift and a floor
-- takes in account the lift configuration and
-- computes the roundtrip distance
distance :: {State,Floor} -> Int;
distance (UP, floor,lo,hi,pending) dest
       = if dest>=floor then
              dest-floor
         else
              2*hi-floor-dest;
distance (WAIT_UP, floor,lo,hi,pending) dest
       = if dest>=floor then
              dest-floor
         else
              2*hi-floor-dest;
distance (DOWN, floor, lo,hi,pending) dest
       = if floor>=dest then
              floor-dest
         else
              dest+floor-2*lo;
distance (WAIT_DOWN, floor, lo,hi,pending) dest
       = if floor>=dest then
              floor-dest
         else
              dest+floor-2*lo;
distance (STOP, floor, lo,hi,pending) dest
       = if floor>=dest then floor-dest else dest-floor;


-- the dispatcher for 2-lifts
box dispatcher
in (s1::State, s2::State, floor::Floor)
out (s1'::State, s2'::State, req::Request)
match
  (s1, s2, floor) ->
        (copy_State s1,
         copy_State s2,
         let d1 = distance s1 floor ;
             d2 = distance s2 floor
         in if d1<=d2 then LIFT1 floor else LIFT2 floor)

| (s1, s2, *) ->
        (copy_State s1, copy_State s2, *)
;


-- single lift controlers
--
box lift1
in (s::State, req::Request)
out (s'::State)
match
   (s, LIFT1 floor) -> (req_lift (copy_State s) floor)
|  (s, *)   ->         (move_lift (copy_State s))
;

box lift2
in (s::State, req::Request)
out (s'::State)
match
  (s, LIFT2 floor) -> (req_lift (copy_State s) floor)
| (s, *)   ->         (move_lift (copy_State s))
;


-- generator for random floor requests
data Maybe a = Nothing | Just a ;

-- generator for a random requests
box generator
in (timer::Maybe Int)
out (timer'::Maybe Int, floor::Int)
match
```

```
  (Nothing) -> (let delay = rem# (abs (randomInt# 0)) 10
                in Just delay,
                rem# (abs (randomInt# 0)) floors)
| (Just t) ->  (decr t, *)
;

decr :: Int -> Maybe Int;
decr n = if n>0 then Just (n-1) else Nothing ;

wire lift1.s  dispatcher.s1';
wire lift1.s' dispatcher.s1;
wire lift1.req dispatcher.req;

wire lift2.s dispatcher.s2';
wire lift2.s' dispatcher.s2;
wire lift2.req dispatcher.req;

wire generator.floor dispatcher.floor;
wire generator.timer generator.timer';

initial generator.timer Nothing;

initial dispatcher.s1 (initialState floors);
initial dispatcher.s2 (initialState floors);
```

B.3 Geometric region server

```
{- A "geometric region server" in Hume inspired by the paper
   "Haskell vs. Ada vs. C++ vs. Awk vs...:
    An Experiment in Software Prototyping Productivity"
   by Paul Hudak and Mark P. Jones -}

type Point = (Float,Float)

-- a geometric region
-- the size measure is the number of inner constructors
data Region^n
         = Left Float                   { n=0 }
         | Right Float                  { n=0 }
         | Below Float                  { n=0 }
         | Above Float                  { n=0 }
         | Circle Point Float           { n=0 }
         | Union Region^p Region^q      { 0<=p, 0<=q, n=1+p+q }
         | Intersect Region^p Region^q  { 0<=p, 0<=q, n=1+p+q }
         | Outside Region^p             { 0<=p, n=1+p }
         ;

-- test if a point is inside a region
```

```
inRegion :: {Point, Region} -> Bool ;

inRegion (x,y) (Left x0)  = x<=.x0 ;
inRegion (x,y) (Right x0) = x0<=.x ;
inRegion (x,y) (Below y0) = y<=.y0 ;
inRegion (x,y) (Above y0) = y0<=.y ;
inRegion (x,y) (Circle (x0,y0) r)
    = let dx = x-.x0 ;
          dy = y-.y0
       in dx*.dx+.dy*.dy <=. r*.r ;

-- intersection, union operators
-- using "if-then-else" to be non-strict on the 2nd argument
inRegion p (Intersect r1 r2)
    = if inRegion p r1 then inRegion p r2 else False ;

inRegion p (Union r1 r2)
    = if inRegion p r1 then True else inRegion p r2 ;

-- complement operator
inRegion p (Outside r) = not (inRegion p r) ;

-- logical negation (not a primitive function)
not :: Bool -> Bool ;
not True = False ;
not False = True;

-- some combinators for more elaborate regions
-- an annular region (r1 < r2)
annulus :: {Point, Float, Float} -> Region ;
annulus p r1 r2 = Intersect (Circle p r2) (Outside (Circle p r1)) ;

-- a vertical strip
vstrip :: {Float,Float} -> Region ;
vstrip x0 x1 = Intersect (Right x0) (Left x1) ;

-- an horizontal strip
hstrip :: {Float,Float} -> Region ;
hstrip y0 y1 = Intersect (Above y0) (Below y1) ;

-- a rectangular region
rect :: {Point, Point} -> Region ;
rect (x0,y0) (x1,y1) = Intersect (vstrip x0 x1) (hstrip y0 y1) ;

-- top half of a circle
halfcircle :: {Point, Float} -> Region ;
halfcircle (x0,y0) r = Intersect (Circle (x0,y0) r) (Above y0) ;

-- copy a 2D-point
-- behaves as the identity function but creates a new heap tuple
```

```
copyPt :: Point -> Point;
copyPt (x,y)= (copyFloat# x, copyFloat# y);

{- the geoserver box
   inputs are: positions of a target and host vessel
   outputs are: boolean flags for zone membership
-}
box geoserver
in (target::Point, host::Point)
out (engage::Bool, weapon::Bool, tight::Bool)
match
(target, host) ->
    (inRegion target (annulus (copyPt host) 10.0 20.0),
     inRegion target (halfcircle (copyPt host) 30.0),
     inRegion target (rect (0.0,0.0) (50.0,10.0)))
;

-- the logger box reports zone intersections
box logger
in (engage::Bool, weapon::Bool, tight::Bool)
out (log::String)
fair
  (True, *, *) -> ("target in engageability zone")
| (False,*, *) -> (*)
| (*, True, *) -> ("target in weapon doctrine zone")
| (*, False,*) -> (*)
| (*, *, True) -> ("target in tight zone")
| (*, *, False) -> (*)
;

-- connect the geoserver and logger together
wire geoserver.engage logger.engage ;
wire geoserver.weapon logger.weapon ;
wire geoserver.tight logger.tight ;
```

Bibliography

Ager, M. S., Biernacki, D., Danvy, O. and Midtgaard, J. (2003*a*), From interpreter to compiler and virtual machine: a functional derivation, Technical Report RS/03/14, BRICS.

Ager, M. S., Biernacki, D., Danvy, O. and Midtgaard, J. (2003*b*), A functional correspondence between evaluators and abstract machines, Technical Report RS/03/13, BRICS.

Amtoft, T., Nielson, F. and Nielson, H. R. (1999), *Type and Effect Systems: Behaviours for Concurrency*, Imperial College Press.

Appel, A. W. (1992), *Compiling with continuations*, Cambridge University Press.

Apple, J. and Weimer, W. (2007), Simulating dependent types with guarded algebraic datatypes. draft paper.

Augustsson, L. (1985), Compiling pattern matching, *in* 'Proceedings of the conference on Functional Programming Languages and Computer Architecture', Springer-Verlag, New York, NY, USA, pp. 368–381.

Augustsson, L. (1998), Cayenne — a language with dependent types, *in* 'Proceddings of the 3rd ACM SIGPLAN International Conference on Functional Programming', ACM, pp. 239–250.

Bagnara, R., Hill, P. M., Ricci, E. and Zaffanella, E. (2003), Precise widening operators for convex polyhedra, *in* R. Cousot, ed., 'Static Analysis: Proceedings of the 10th International Symposium', Vol. 2694 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, San Diego, California, USA, pp. 337–354.
URL: *http://www.cs.unipr.it/ppl/Documentation/BagnaraHRZ03.pdf*

Bagnara, R., Hill, P. M. and Zaffanella, E. (2002), A new encoding of not necessarily closed convex polyhedra, *in* M. Carro, C. Vacheret and K.-K. Lau, eds, 'Proceedings of the 1st CoLogNet Workshop on Component-based Software Development and Implementation Technology for Computational Logic Systems',

Madrid, Spain, pp. 147–153. Published as TR Number CLIP4/02.0, Universidad Politécnica de Madrid, Facultad de Informática.
URL: *http://www.cs.unipr.it/ppl/Documentation/BagnaraHZ02a.pdf*

Bagnara, R., Hill, P. M. and Zaffanella, E. (2003), Widening operators for powerset domains, *in* B. Steffen and G. Levi, eds, 'Verification, Model Checking and Abstract Interpretation: Proceedings of the 5th International Conference (VMCAI 2004)', Vol. 2937 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Venice, Italy, pp. 135–148.
URL: *http://www.cs.unipr.it/ppl/Documentation/BagnaraHZ03b.pdf*

Bagnara, R., Hill, P. M. and Zaffanella, E. (2006), The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems, Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.MS/0612085`, available from `http://arxiv.org/`.

Bagnara, R., Ricci, E., Zaffanella, E. and Hill, P. M. (2002), Possibly not closed convex polyhedra and the Parma Polyhedra Library, *in* M. V. Hermenegildo and G. Puebla, eds, 'Static Analysis: Proceedings of the 9th International Symposium', Vol. 2477 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Madrid, Spain, pp. 213–229.
URL: *http://www.cs.unipr.it/ppl/Documentation/BagnaraRZH02.pdf*

Bagnara, R., Rodríguez-Carbonell, E. and Zaffanella, E. (2005), Generation of basic semi-algebraic invariants using convex polyhedra, *in* C. Hankin, ed., 'Static Analysis: Proceedings of the 12th International Symposium', Lecture Notes in Computer Science, Springer-Verlag, Berlin, London, UK.

Bakewell, A. (2001), An Operational Theory of Relative Space Efficiency, PhD thesis, Department of Computer Science, University of York.

Barwise, J. and Moss, L. (1996), *Vicious Circles*, Center for the Study of Language and Information, Leland Standford Junior University, chapter 17.

Benton, N., Hughes, J. and Moggi, E. (2000), 'Monads and effects', Lecture Notes of the Applied Semantics Summer School.

Bird, R. and Wadler, P. (1988), *Introduction to Functional Programming*, Prentice Hall International.

Birkedal, L., Tofte, M. and Vejlstrup, M. (1996), From region inference to von Neumann machines via region representation inference, *in* 'Proceedings of the 23rd

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', pp. 171–183.

Bjerner, B. and Holmström, S. (1989), A compositional approach to time analysis of first order lazy functional programs, *in* 'Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture', ACM Press, New York, NY, USA, pp. 157–165.

Bonenfant, A., Ferdinand, C., Hammond, K. and Heckman, R. (2007), Worst-case execution times for a purely functional language, *in* 'Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages', Vol. 4449 of *Lecture Notes in Computer Science*, Springer.

Brady, E. and Hammond, K. (2006), A dependently typed framework for static analysis of program execution costs, *in* 'Implementation of Functional Languages (IFL) 2005', Vol. 4015 of *Lecture Notes in Computer Science*, Springer, Berlin/Heidelberg, pp. 74–90.

Burns, A. and Wellings, A. (1996), *Real-Time Systems and Programming Languages*, second edn, Addison-Wesley.

Campbell, B. (2008), Type-based amortized stack memory prediction, PhD thesis, University of Edinburgh.

Chakravarty, M. M. T. and Keller, G. C. (2002), *An Introduction to Computing (with Haskell)*, Pearson Education Australia.

Chandru, V. (1993), 'Variable elimination in linear constraints', *The Computer Journal* **36**(5), 463–472.

Chernikova, N. V. (1968), 'Algorithm for discovering the set of all solutions of a linear programming problem', *U.S.S.R. Computational Mathematics and Mathematical Physics* **8**(6), 282–293.

Chin, W.-N. and Khoo, S.-C. (2001), 'Calculating sized types', *Higher Order and Symbolic Computation* **14**(2-3), 261–300.

Chin, W.-N., Khoo, S.-C. and Xu, D. N. (2003), Extending sized type with collection analysis, *in* 'Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation (PEPM'03)', ACM Press, New York, NY, USA, pp. 75–84.

Cooper, D. C. (1972), Theorem proving in arithmetic without multiplication, *in* B. Meltzer and D. Michie, eds, 'Machine Intelligence', Edinburgh University Press, pp. 91–100.

Coq (2006), *The Coq proof assistant — reference manual*, 8.1 edn.

Coquand, C. and Coquand, T. (1999), Structured type theory, *in* 'Proceedings of the Workshop on Logical Frameworks and Meta-languages'.
URL: *citeseer.ist.psu.edu/article/coquand99structured.html*

Cousot, P. and Cousot, R. (1976), Static determination of dynamic properties of programs, *in* 'Proceedings of the 2nd International Symposium on Programming', Dunod, Paris, France, pp. 106–130.

Cousot, P. and Cousot, R. (1977), Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *in* 'Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)', ACM Press, New York, NY, Los Angeles, California, pp. 238–252.

Cousot, P. and Cousot, R. (1992*a*), 'Abstract interpretation frameworks', *Journal of Logic and Computation* **2**(4), 511–547.

Cousot, P. and Cousot, R. (1992*b*), Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper, *in* M. Bruynooghe and M. Wirsing, eds, 'Proceedings of the International Workshop Programming Language Implementation and Logic Programming (PLILP'92)', Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, Springer-Verlag, Berlin, Germany, pp. 269–295.

Cousot, P. and Halbwachs, N. (1978), Automatic discovery of linear restraints among variables of a program, *in* 'Proceedings of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'78)', ACM Press, New York, NY, Tucson, Arizona, pp. 84–97.

Crary, K. and Weirich, S. (2000), Resource bound certification, *in* 'Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)', New York, NY, USA, pp. 184–198.

Dal Lago, U. and Martini, S. (2005), 'An invariant cost model for the lambda calculus', arXiv:cs.LO/0511045v1.

Damas, L. (1985), Type Assignment in Programming Languages, PhD thesis, Department of Computer Science, University of Edinburgh.

Damas, L. and Milner, R. (1982), Principal type-schemes for functional programs, *in* 'Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82)', pp. 207–212.

Danielsson, N. A. (2008), Lightweight semiformal time complexity analysis for purely functional data structures, *in* 'Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)'.

Danvy, O. (2003), A rational deconstruction of the SECD machine, Technical Report RS/03/33, BRICS.

Danvy, O. and Nielsen, L. R. (2001), Defunctionalization at work, *in* 'PPDP '01: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming', ACM Press, New York, NY, USA, pp. 162–174.

Davey, B. A. and Priestley, H. A. (1990), *Introduction to Lattices and Order*, Cambridge Mathematical Textbooks.

Dijsktra, E. W. (1970), Notes on structured programming, Technical Report 70-WSK-03, Technological University Eindhoven, Department of Mathematics, The Netherlands.

Dornic, V., Jouvelot, P. and Gifford, D. K. (1992), 'Polymorphic time systems for estimating program complexity', *ACM Letters on Programming Languages and Systems (LOPLAS)* **1**(1), 33–45.
URL: *http://www.psrg.lcs.mit.edu/ftpdir/papers/loplas.dvi*

Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S. and Wilhelm, R. (2001), Reliable and precise WCET determination for a real-life processor, *in* 'Proceedings of the First International Workshop on Embedded Software', Springer-Verlag, London, UK, pp. 469–485.

Ferdinand, C., Martin, F., Wilhelm, R. and Alt, M. (1999), 'Cache behavior prediction by abstract interpretation', *Science of Computerd. Programming* **35**(2-3), 163–189.

Finnie, G. and Amey, P. (2006), *SPARK 95 — The SPADE Ada 95 Kernel (excluding RavenSPARK)*, 4.7 edn, Praxis High Integrity Systems.

Fuh, Y.-C. and Mishra, P. (1988), Type inference with subtypes, *in* 'Proceedings of the 2nd European Symposium on Programming', North-Holland, pp. 155–175.

Fuh, Y.-C. and Mishra, P. (1989), Polymorphic subtype inference: closing the theory-practice gap, *in* 'Proceedings of the International Joint Conference on Theory and Practice of Software Development', number 352 *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 167–183.

Girard, J.-Y., Taylor, P. and Lafont, Y. (1989), *Proofs and Types*, Cambridge University Press.

Grobauer, B. (2001), Cost recurrences for DML programs, *in* 'Proceedings of the 6th ACM SIGPLAN international conference on Functional programming', ACM Press, New York, NY, USA, pp. 253–264.

Guibas, L. and Sedgewick, R. (1978), A dichromatic framework for balanced trees, *in* 'IEEE Symposium on Foundations of Computer Science', pp. 8–21.

Halbwachs, N. (1979), Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme, PhD thesis, Université Scientifique et Médicale de Grenoble.

Hammond, K. (2003), 'An abstract machine implementation for hume'.
URL: *http://www-fp.cs.st-andrews.ac.uk/hume/papers/HAM.ps*

Hammond, K. and Michaelson, G. (2002), Predictable space behaviour in FSM-Hume, *in* R. Peña and T. Arts, eds, 'Implementation of Functional Languages: 14th International Workshop, IFL 2002, Madrid, Spain, September 16-18, 2002, Revised Selected Papers', Vol. 2670 of *Lecture Notes in Computer Science*, Springer, pp. 1–16.

Hammond, K., Michaelson, G. and Pointon, R. (2007), 'The Hume report (version 1.1)'.
URL: *http://www-fp.cs.st-andrews.ac.uk/hume/report/hume-report.pdf*

Henglein, F., Makholm, H. and Niss, H. (2001), A direct approach to control-flow sensitive region-based memory management, *in* 'Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming', ACM, Montréal, Canada, pp. 175–186.
URL: *citeseer.ist.psu.edu/henglein01direct.html*

Hofmann, M. (2000), 'A type system for bounded space and functional in-place update', *Nordic Journal of Computing* **7**(4), 258–289.

Hofmann, M. (2002), The strength of non-size increasing computation, *in* 'Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)', ACM, New York, NY, USA, pp. 260–269.

Hofmann, M. and Jost, S. (2003), Static prediction of heap space usage for first-order functional programs, *in* 'Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)', pp. 185–197.

Hofmann, M. and Jost, S. (2006), Type-based amortised heap-space analysis, *in* P. Sestoft, ed., 'Proceedings of the 15th European Symposium on Programming (ESOP'06)', Vol. 3924 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 22–37.

Hope, C. and Hutton, G. (2006), Accurate Step Counting, *in* 'Implementation and Application of Functional Languages', Vol. 4015 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg. Selected papers from the 17th International Workshop on Implementation and Application of Functional Languages, Dublin, Ireland, September 2005.

Hudak, P., Hughes, J., Jones, S. P. and Wadler, P. (2007), A history of Haskell: being lazy with class, *in* 'Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III)', ACM, New York, NY, USA, pp. 12–1–12–55.

Hudak, P. and Jones, M. P. (1994), 'Haskell vs. Ada vs. C++ vs Awk vs . . . an experiment in software prototyping productivity'.
URL: *citeseer.ist.psu.edu/hudak94haskell.html*

Hughes, J. (1989), 'Why functional programming matters', *Computer Journal* **32**(2), 98–107.

Hughes, J. and Pareto, L. (1999), Recursion and dynamic data-structures in bounded space: Towards embedded ML programming, *in* 'Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99)', Vol. 34, pp. 70–81.

Hughes, J., Pareto, L. and Sabry, A. (1996), Proving the correctness of reactive systems using sized types, *in* G. L. S. Jr, ed., 'Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)', Vol. 23, ACM, St Petersburg, Florida.

Hutton, G. (2007), *Programming in Haskell*, Cambridge University Press.

Jones, M. P. (2000), Type classes with functional dependencies, *in* 'Proceedings of the 9th European Symposium on Programming Languages and Systems', Vol. 1782, pp. 230–244.
URL: *citeseer.ist.psu.edu/jones00type.html*

Jones, R. and Lins, R. D. (1996), *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons.

Jones, S. L. P. (1992), 'Implementing lazy functional languages on stock hardware: the spineless tagless G-machine version 2.5', *Journal of Functional Programming* **2**(2), 127–202.

Jones, S. L. P. and Johnsson, T. (1987), Optimizing generalized tail calls, *in* S. L. P. Jones, ed., 'The Implementation of Functional Programming Languages', Prentice-Hall International, chapter 21, pp. 367–379.

Jones, S. P., Eber, J.-M. and Seward, J. (2000), Composing contracts: an adventure in financial engineering (functional pearl), *in* 'Proceedings of the 5th ACM SIGPLAN international conference on Functional programming', ACM, New York, NY, USA, pp. 280–292.

Jones, S. P., ed. (2003), *Haskell 98 Language and Libraries — The Revised Report*, Cambridge University Press.
URL: *http:://www.haskell.org*

Jones, S. P., Vytiniostis, D., Weirich, S. and Washburn, G. (2006), Simple unification-based type inferece for GADTs, *in* 'Proceedings of the International Conference on Functional Programming'.

Jouvelot, P. and Gifford, D. K. (1991), Algebraic reconstruction of types and effects, *in* 'Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'91)', pp. 303–310.
URL: *http://www.psrg.lcs.mit.edu/ftpdir/papers/popl91.dvi*

Kapur, D. (2004), Automatically generating loop invariants using quantifier elimination, *in* 'IMACS International Conference on Applications of Computer Algebra', Beaumont, Texas.

Kelly, W., Pugh, W., Rosser, E. and Shpeisman, T. (1996), Transitive closure of infinite graphs and its applications, *in* 'Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing', Springer-Verlag, pp. 126–140.
URL: *citeseer.ist.psu.edu/kelly95transitive.html*

Khachiyan, L., Boros, E., Borys, K., Elbassioni, K. and Gurvich, V. (2006), Generating all vertices of a polyhedron is hard, *in* 'Proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm', ACM, New York, NY, USA, pp. 758–765.

Knuth, D. E. (1973), *The Art of Computer Programming*, Vol. 1: Fundamental algorithms, second edn, Addison-Wesley.

Kogge, P. M. (1991), *The Architecture of Symbolic Computers*, McGraw Hill International Editions.

Koubarakis, M. (2006), Temporal CSPs, *in* F. Rossi, P. van Beek and T. Walsh, eds, 'Handbook of Constraint Programming', Elsevier, chapter 19, pp. 683–685.

Lacan, P., Monfort, J. N., Ribal, L. V. Q., Deutsch, A. and Gonthier, G. (1998), ARIANE 5 — the software reliability verification process: The ARIANE 5 example, *in* B. Kaldeich-Schürmann, ed., 'Proceedings of the DASIA'98 Conference on 'Data Systems in Aerospace'', European Space Agency, pp. 201–205.

Landin, P. (1964), 'The mechanical evaluation of expressions', *The Computer Journal* **6**(4), 308–320.

Le Métayer, D. (1988), 'Ace: An automatic complexity evaluator', *ACM Transactions on Programming Languages and Systems (TOPLAS)* **10**(2), 248–266.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D. and Vouillon, J. (2007), *The Objective Caml System release 3.10: Documentation and user's manual*, INRIA.
URL: *http://caml.inria.fr/*

Lewis, H. R. and Papadimitriou, C. H. (1981), *Elements of the theory of computation*, Prentice-Hall International.

Liu, Y. A. and Gomez, G. (1998), Automatic accurate time-bound analysis for high-level languages, *in* 'Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)', Springer-Verlag, London, UK, pp. 31–40.

Loidl, H.-W. (1998), Granularity in Large-Scale Parallel Functional Programming, PhD thesis, Department of Computing Science, University of Glasgow.
URL: *http://www.cee.hw.ac.uk/˜dsg/gph/papers/ps/loidl-thesis.ps.gz*

Lucassen, J. M. (1987), Types and Effects: Towards the Integration of Functional an Imperative Programming, PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science.

Lucassen, J. M. and Gifford, D. K. (1988), Polymorphic effect systems, *in* 'Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)', ACM, New York, NY, USA, pp. 47–57.

Luo, Z. and Pollack, R. (1992), Lego proof development system: User's manual, Technical report, Department of Computer Science, University of Edinburgh.

MacQueen, D. B. and Sethi, R. (1982), A semantic model of types for applicative languages, *in* 'Proceedings of the ACM symposium on LISP and functional programming', ACM, New York, NY, USA, pp. 243–252.

Manna, Z. (1974), *Mathematical theory of computation*, McGraw Hill.

McBride, C. (2002), 'Faking it: simulating dependent types in Haskell', *Journal of Functional Programmming* **12**(5), 375–392.

McBride, C. and McKinna, J. (2004), 'The view from the left', *Journal of Functional Programming* **14**(1), 69–111.

Meyer, A. R. and Ritchie, D. M. (1967), The complexity of loop programs, *in* 'Proceedings of the 1967 22nd national conference', ACM, New York, NY, USA, pp. 465–469.

Michaelson, G., Hammond, K. and Sérot, J. (2005), FSM-Hume is finite state, *in* S. Gilmore, ed., 'Trends in Functional Programming', Vol. 4, Intellect, pp. 19–28.
URL: *http://homepages.inf.ed.ac.uk/stg/workshops/TFP/book/*

Milner, R. (1976), 'A theory of type polymorphism in programming', *Journal of Computer System Sciences* **17**(3), 348–375.

Milner, R., Tofte, M., Harper, R. and MacQueen, D. (1997), *The Definition of Standard ML (revised)*, MIT Press.

Minsky, M. (1967), *Computation: Finite and Infinite Machines*, Prentice Hall.

Mitchell, J. C. (1984), Coercion and type inference, *in* 'Proceedings of the 11th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'84)', ACM, New York, NY, USA, pp. 175–185.

Mitchell, J. C. (1996), *Foundations for Programming Languages*, MIT Press.

Motzkin, T. S., Raiffa, H., Thompson, G. L. and Thrall, R. M. (1953), The double description method, *in* H. W. Kuhn and A. W. Tucker, eds, 'Contributions to the Theory of Games – Volume II', number 28 *in* 'Annals of Mathematics Studies', Princeton University Press, Princeton, New Jersey, pp. 51–73.

Necula, G. C. (1997), Proof-carrying code, *in* 'Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)', New York, NY, USA, pp. 106–119.

Nielson, F. and Nielson, H. R. (1999), Type and effect systems, *in* 'Correct System Design', pp. 114–136.
URL: *citeseer.ist.psu.edu/nielson99type.html*

Nielson, F., Nielson, H. R. and Amtoft, T. (1996*a*), Polymorphic subtyping for effect analysis: The algorithm, *in* 'Logical and Operational Methods in the Analysis of Programs and Systems', pp. 207–243.
URL: *citeseer.ist.psu.edu/article/nielson97polymorphic.html*

Nielson, F., Nielson, H. R. and Hankin, C. (1999), *Principles of Program Analysis*, Springer.

Nielson, H. R., Nielson, F. and Amtoft, T. (1996*b*), Polymorphic subtyping for effect analysis: The static semantics, *in* 'Logical and Operational Methods in the Analysis of Programs and Systems', pp. 141–171.
URL: *citeseer.ist.psu.edu/article/nielson97polymorphic.html*

Okasaki, C. (1998), *Purely Functional Data Structures*, Cambridge University Press.

Owicki, S. and Lamport, L. (1982), 'Proving liveness properties of concurrent programs', *ACM Transactions on Programming Languages and Systems* **4**(3), 455–495.

Pareto, L. (1998), 'Sized types', Dissertation for the Licentiate Degree in Computing Science, Department of Computing Science, Chalmers University of Technology. ISBN 91-7197-682-5.

Pareto, L. (2000), Types for Crash Prevention, PhD thesis, Chalmers University of Technology, Göteborg.

Pierce, B. C. (2002), *Types and Programming Languages*, The MIT Press.

Plotkin, G. D. (1981), A structural approach to operational semantics, Technical Report DAIMI FN-19, University of Aarhus.

Pohlmann, K. C. (1989), *Principles of Digital Audio*, second edn, SAMS, Carmel, Indiana, USA.

Portillo, A. R., Hammond, K., Loidl, H.-W. and Vasconcelos, P. (2003), Cost analysis using automatic size and time inference, *in* 'Proceedings of the 14th International Workshop on Implementation of Functional Languages', Vol. 2670 of *LNCS*, Springer-Verlag.

Pugh, W. (1992), 'The Omega test: A fast and practical integer programming algorithm for dependence analysis', *Communications of the ACM* **8**, 102–114.

Rabin, M. (1977), Decidable theories, *in* J. Barwise, ed., 'Handbook of Mathematical Logic', North Holland, chapter 3, pp. 600–606.

Reingold, E. M. (1973), 'A nonrecursive list moving algorithm', *Communications of the ACM* **16**(5), 305–307.

Reistad, B. and Gifford, D. K. (1994), Static Dependent Costs for Estimating Execution Time, *in* 'Proceedings of the 1994 ACM Conference on Lisp and Functional Programming – LFP '94', Orlando, FL, pp. 65–78.

Reynolds, J. C. (1972), Definitional interpreters for higher-order programming languages, *in* 'ACM '72: Proceedings of the ACM annual conference', ACM Press, New York, NY, USA, pp. 717–740.

Robinson, J. A. (1971), 'Computational logic: The unification computation', *Machine Inteligence* **6**, 63–72.

Rosendahl, M. (1989), Automatic complexity analysis, *in* 'Proceedings of the 1989 International Conference on Functional Programmings Languages and Computer Architecture (FPCA'89)', pp. 144–156.

Runciman, C. and Wakeling, D. (1993), 'Heap profiling of lazy functional programs', *Journal of Functional Programming* **3**(2), 217–245.
URL: *citeseer.ist.psu.edu/article/runciman93heap.html*

Samson, P. M. and Jones, S. L. P. (1995), Time and space profiling for non-strict, higher-order functional languages, *in* 'Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)', San Francisco, California, pp. 355–366.

Sands, D. (1990), Calculi for Time Analysis of Functional Programs, PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London.

Schrijver, A. (1986), *Theory of Linear and Integer Programming*, Wiley Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons.

Simões, H., Hammond, K., Florido, M. and Vasconcelos, P. (2007), Using intersection types for cost-analysis of higher-order polymorphic functional programs, *in* 'Proceedings of the Types Project', Vol. 4502 of *Lecture Notes in Computer Science*, Springer.

Smullyan, R. (1995), *First-Order Logic*, Dover.

SPARK Team (2005), RavenSPARK design for the mine pump case study, Technical Report S.P0468.76.21, Praxis High Integrity Systems.
URL: *http://www.praxis-his.com/sparkada/pdfs/minepump.pdf*

Stankovic, J. A. (1988), ‘Misconceptions about real-time computing: A serious problem for next-generation systems’, *Computer* **21**(10), 10–19.

Steele, G. L. (1977), Debunking the ‘expensive procedure call’ myth, or, procedure call implementations considered harmful, or, lambda: the ultimate GOTO, Technical Report AI Lab Memo AIM-443, MIT AI Lab.
URL: *http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-443.pdf*

Stoy, J. E. (1977), *Denotational Semantics: the Scott-Strachey approach to programming language theory*, The MIT Press.

Talpin, J.-P. and Jouvelot, P. (1992), ‘Polymorphic type, region and effect inference’, *Journal of Functional Programming* **2**(3), 245–271.

Talpin, J.-P. and Jouvelot, P. (1994), ‘The type and effect discipline’, *Information and Computation* **111**(2), 245–296.

Tarjan, R. E. (1985), ‘Amortized computational complexity’, *SIAM Journal on Algebraic and Discrete Methods* **6**(2), 306–318.

Tofte, M. (1988), Operational Semantics and Polymorphic Type Inference, PhD thesis, University of Edinburgh.

Tofte, M. and Birkedal, L. (1998), ‘A region inference algorithm’, *ACM Transactions on Programming Languages and Systems* **20**(4), 724–767.
URL: *citeseer.ist.psu.edu/tofte98region.html*

Tofte, M., Birkedal, L., Elsman, M. and Hallenberg, N. (2004), ‘A retrospective on region-based memory management’, *Higher-Order and Symbolic Computation* **17**(3), 245–265.

Tofte, M. and Talpin, J.-P. (1997), ‘Region-based memory management’, *Information and Computation* .
URL: *citeseer.ist.psu.edu/tofte97regionbased.html*

Tolmach, A. (2001), ‘An external representation for the GHC core language’.
URL: *citeseer.ist.psu.edu/tolmach01external.html*

Turner, D. A. (1995), Elementary strong functional programming, *in* ‘Proceedings of the First International Symposium on Functional Programming Languages in Education’, Springer-Verlag, London, UK, pp. 1–13.

Turner, D. A. (2004), ‘Total functional programming’, *Journal of Universal Computer Science* **10**(7), 751–768.

Unnikrishnan, L., Stoller, S. D. and Liu, Y. A. (2000), Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages, Technical Report 538, Computer Science Dept, Indiana University.

Vasconcelos, P. and Hammond, K. (2004), Inferring cost equations for recursive, higher-order and polymorphic functional programs, *in* 'Proceedings of the 14th International Workshop on Implementation of Functional Languages', Vol. 3145 of *LNCS*, Springer-Verlag.

Verge, H. L. (1992), A note on Chernikova's algorithm, *Publication interne* 635, IRISA, Campus de Beaulieu, Rennes, France.

Wadler, P. (1987), Efficient compilation of pattern matching, *in* S. L. P. Jones, ed., 'The Implementation of Functional Programming Languages', Prentice-Hall International, chapter 5, pp. 396–408.

Wadler, P. (1988), Strictness analysis aids time analysis, *in* 'Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)', pp. 119–132.

Wadler, P. (1993), Monads for functional programming, *in* M. Broy, ed., 'Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School', Springer-Verlag.
URL: *citeseer.ist.psu.edu/wadler95monads.html*

Wadler, P. (1998*a*), The marriage of effects and monads, *in* 'Proceddings of the International Conference on Functional Programming', Baltimore.

Wadler, P. (1998*b*), 'Why no one uses functional languages', *SIGPLAN Notices* **33**(8), 23–27.
URL: *citeseer.ist.psu.edu/wadler98why.html*

Wadler, P. and Blott, S. (1989), How to make *ah-hoc* polymorphism less *ad hoc*, *in* 'Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89)', ACM, pp. 60–76.

Wegbreit, B. (1975), 'Mechanical program analysis', *Communications of the ACM* **18**(9), 528–539.

Wilde, D. K. (1993), A library for doing polyhedral operations, Master's thesis, Oregon State University, Corvallis, Oregon. Also published as IRISA *Publication interne* 785, Rennes, France, 1993.
URL: *ftp://ftp.ee.byu.edu/faculty/Wilde/Polylib/report.ps.gz*

Wilson, P. R., Johnstone, M. S., Neely, M. and Boles, D. (1995), Dynamic storage allocation: a survey and critical review, *in* 'Proceedings of the 1995 International Workshop on Memory Managment', Lecture Notes in Computer Science, Springer Verlag.

Winskel, G. (1993), *The Formal Semantics of Programming Languages*, The MIT Press.

Wright, A. K. (1995), 'Simple imperative polymorphism', *Lisp and Symbolic Computation* **8**(4), 343–355.
URL: *citeseer.ist.psu.edu/wright95simple.html*

Xi, H. (1998), Dependent Types in Practical Programming, PhD thesis, Carnegie Mellon University.

Xi, H. and Pfenning, F. (1999), Dependent types in practical programming, *in* 'Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)', pp. 214–227.