Imperial College of Science, Technology and Medicine
Department of Computing

Late-bound code generation

Matthew Robert Taylor

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London, November 2022

Abstract

Each time a function or method is invoked during the execution of a program, a stream of instructions is issued to some underlying hardware platform. But exactly *what* underlying hardware, and *which* instructions, is usually left implicit. However in certain situations it becomes important to control these decisions. For example, particular problems can only be solved in real-time when scheduled on specialised accelerators, such as graphics coprocessors or computing clusters.

We introduce a novel operator for hygienically reifying the behaviour of a runtime function instance as a syntactic fragment, in a language which may in general differ from the source function definition. Translation and optimisation are performed by recursively invoked, dynamically dispatched code generators. Side-effecting operations are permitted, and their ordering is preserved.

We compare our operator with other techniques for pragmatic control, observing that: the use of our operator supports lifting arbitrary mutable objects, and neither requires rewriting sections of the source program in a multi-level language, nor interferes with the interface to individual software components. Due to its lack of interference at the abstraction level at which software is composed, we believe that our approach poses a significantly lower barrier to practical adoption than current methods.

The practical efficacy of our operator is demonstrated by using it to offload the user interface rendering of a smartphone application to an FPGA coprocessor, including both statically and procedurally defined user interface components. The generated pipeline is an application-specific, statically scheduled processor-per-primitive rendering pipeline, suitable for place-and-route style optimisation.

To demonstrate the compatibility of our operator with existing languages, we show how it may be defined within the Python programming language. We introduce a transformation for weakening mutable to immutable named bindings, termed *let-weakening*, to solve the problem of propagating information pertaining to named variables between modular code generating units.

Acknowledgements

I would like to thank my supervisor Professor Paul H J Kelly, for his seemingly endless patience and good nature, and my loving wife Virginie Taylor (née Serneels) for her constant kindness and encouragement. Immense thanks also goes to my parents Caroline and Glynn Taylor, for their abiding love and support. I am also deeply grateful to my friends, with whom I have had many helpful and thought-provoking conversations (alphabetically): Sean Bremner, Eduardo Da Costa Carvalho, David Drakard, Tianjiao Sun and Emanuele Vespa.

Dedication

```
'''(lambda c: eval(c))(\'\'\'(lambda c, _:
"(lambda c: eval(c))(" + "'"*3 + c[:158] +
'%02d'%((int(c[158:160])>>1)|(8 if (int(c[
158:160])+1)//2%2 else 0)) + c[160:] + "'"
*3 + ")")(c, # 04
(lambda o, b7f4b4d7e9eaa09f9eeff7daf1ebe6:
print(chr(o) if (o>>4)!=3 else chr(128582+
o*-9), end=''))((int(c[170+2*int(c[158:160
]):172+2*int(c[158:160])], 16) + sum([ord(
a) for a in c[:172]]))&0xff, 0))\'\'\')'''
```

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Whilst all imperative programming languages have an operator for *invoking* functions, not all have an operator for *analysing* functions. We describe and demonstrate a novel operator acting upon a runtime function instance (closure) which allows the behaviour of its operand to be examined, optimised, and deployed for execution on heterogeneous local or non-local hardware.

Each time a function or method is invoked during the execution of a program, a stream of instructions is issued to some underlying hardware platform. But exactly *what* underlying hardware, and *which* instructions, is usually left implicit. Such details are typically abstracted by the programming language. However in certain situations it becomes important to control these decisions—to affect *pragmatic control*. For example particular problems can only be solved in real-time when scheduled on specialised accelerators, such as graphics coprocessors or computing clusters. Yet it is generally not possible for a running program to exert dynamic control over the execution policy of the function's invocation operator itself. Specifically by the term "pragmatic control" we mean the control of aspects of a program's execution which do not bear on the correctness of the values which it produces: for instance controlling which hardware unit is used to execute a particular section of the program, which machine instructions are used to implement a particular operation, or how instructions are scheduled in time.

Various solutions to this problem have been formulated, many of which rely upon obtaining some reified representation of the function. This might be accomplished by (i) augmenting the programming language with additional introspection and reification operators, (ii) modifying the function definition such that it is complicit in disclosing its behaviour, for instance returning a representation of its implementation when invoked, or (iii) implementing the function non-natively, for instance as a data structure. By the term "reification" we mean to make some aspect of a program's behaviour or state accessible to the program itself, typically as a runtime value. For example, a section of a program's behaviour might be reified as list of machine instructions, or the state of a particular scope might be reified as a dictionary from strings to values. Once an explicit representation of the function's behaviour has been obtained, it may be programmatically computed upon, analysed, translated and issued for execution on arbitrary hardware.

In general, solutions of form (ii) and (iii) suffer from the major disadvantage that the syntax of the analysed function definition must be modified, or at the very least its type signature or interface definition must be changed. These modifications are usually peculiar to the analysis framework used, and so pose serious challenges to reuse and integration of software components which have been modified in this way. For these reasons, we take the approach of form (i). Traditional solutions of form (i) typically focus on reification of the function definition's source syntax, although facilities for scope reification are not uncommon. Whilst in theory reifying both a function's source syntax and closure scope provide a sufficient foundation for solving the problem as stated, there is considerable complexity involved in generating optimised or translated forms of the reified function when starting from this basis.

We introduce our novel operator for late-bound code generation, named "`translate`", which in contrast to conventional operators for syntactic reification: (a) returns a fragment which in general differs from the syntax of the source function definition, whereby all names are hygienically bound via explicit reference to the closing scope, or otherwise inlined; (b) returns a fragment in any arbitrary language, not necessarily the language of the source function definition, such that encapsulation of the source implementation may be preserved; (c) constructs the resulting fragment by means of dynamic dispatch, delegating preferentially to (overridable) code generators associated directly with the function instance, its type, or other related metadata. For a glossary of related terms, see Appendix A. From a technical perspective, compared to previous approaches of form (i)—such as Template Haskell [SJ02]—`translate` supports "lifting" a broader class of objects into generated code, including arbitrary mutable objects.

Unlike solutions of form (ii) and (iii), using `translate` neither requires rewriting sections of the source program in a multi-level language, nor interferes with the interface to individual software components. Due to its lack of interference at the abstraction level at which software is composed, we believe `translate` poses a significantly lower barrier to practical adoption in situations where pragmatic control over the execution of a program is required.

1.2 Contributions

- We describe "late-bound code generation", our novel approach to code generation, capable of translating the behaviour of side-effecting function instances at runtime for execution on heterogeneous targets, which neither requires the function to be implemented in a multi-level language, nor interferes with the function interface or type-signature.

- We demonstrate late-bound code generation by applying it to the problem of offloading the rasterisation of a smartphone user interface to an FPGA coprocessor. This consists of two independent sub-contributions:

  – First, we define an application-specific object-processor pipeline rendering architecture, suitable for execution within an FPGA coprocessor.

  – Second, we apply our code generation system to a source program written in terms of contemporary user interface abstractions, offloading its rasterisation according to our architecture, and featuring both standard and programmatically defined user interface components.

- We show how `translate`, our operator for late-bound code generation, may be implemented in the Python programming language. We introduce a transformation for weakening mutable to immutable named bindings, termed *let-weakening*, to solve the problem of propagating information related to named variables between code generators.

1.3 Structure

The order of Chapters 2 and 3 within this thesis departs from convention: our discussion of the relevant literature is delayed until Chapter 3, and is preceded by a characterisation of our approach in Chapter 2. This departure was intentional, being motivated by the need to clarify the topic of the thesis, before and in order to properly frame our discussion of the related work. A brief summary of the structure of the thesis follows.

- In Chapter 2 we characterise "late-bound code generation", our novel approach to code generation. We start by introducing the fundamental concept underlying late-bound code generation, termed "syntactification". Immediately following this broad, abstract description of our approach, we narrow our focus in order to be more concrete: describing the particulars of its central `translate` operator. To motivate our discussion we give a simple, but non-trivial, application of `translate`—using it to offload work from an interpreter for the Python programming language to an interpreter for the Scheme programming language. A "toy" implementation for the `translate` operator in Python is sketched, in order to address certain key concepts—in Chapter 5 we return to the question of how `translate` might be integrated into Python, in detail.
- In Chapter 3 we survey the literature which relates most closely to late-bound code generation. Broadly speaking we limit our discussion to approaches whereby control over where, when and how a program's computation is performed is encoded within the program itself, rather than within some external system. We characterise the related work into one of three informal categories, specifically based upon: directives, reification, and explicit construction of target syntax. Late-bound code generation most closely aligns with other approaches based upon reification, and so we explore that category in the greatest depth. In particular, we discuss the closely related languages Template Haskell

[SJ02] and Converge [Tra05] in section 3.3.4, and how our approach goes beyond their existing capabilities.

- In Chapter 4 we test the practicality of our approach by using it to offload the rendering of a smartphone user interface to an FPGA coprocessor—the objective being to reduce the energy consumed due to graphical rasterisation. Although our primary motivation behind this experiment is to assess if our approach is capable of accomplishing the task (whether or not our energy objective is met) we detail the various opportunities for reducing energy expenditure, and evaluate our experimental results. Our reflections on this experiment focus on the degree to which our application's source code either had to be modified or annotated, in order to achieve a feasible circuit.

- In Chapter 5 we demonstrate the possibility of performing late-bound code generation in a general purpose programming language, by designing and implementing ReiPy: a late-bound code generation system for Python. We build upon Python's inherent introspection facilities, in order to simulate the `translate` operator as an ordinary function, without any need to modify the language or interpreter itself. In this chapter we only consider Python-to-Python translations, and do not attempt to translate the full language. Some omissions include the lack of support for: early function return; throwing or catching exceptions; coroutines or generators; variable shadowing; and non-positional arguments. One notable challenge to supporting late-bound code generation in a language with mutable variable bindings, is in propagating static information between code generators corresponding to the definition and the use of named variables. We introduce a solution to this problem, which selectively weakens assignments to let-bindings: a transformation which we call "let-weakening".

- In Chapter 6 we critique some of the theoretical and technical shortcomings of late-bound code generation. We also include a detailed comparison of our approach with the closely related system Lancet, described by Rompf et al. [RSB$^+$14]. We delay our comparison with Lancet to this final chapter, since it involves a number of technical details which have only been established by this point in the thesis. Finally, we suggest a number of interesting directions for future work: in particular, we suggest replacing ReiPy's current direct style metacircular interpreter with a continuation-passing style interpreter, in order to support translating terms in the full Python language.

1.4 Statement of originality

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

1.5 Copyright declaration

Chapter 2

Late-bound code generation

2.1 Introduction

Late-bound code generation, the primary contribution of this thesis, is a runtime technique for translating the behaviour of function instances (closures) to code fragments in some specified target language. It builds upon our underlying approach to translation, which we term "syntactification"—see Appendix A for a glossary of related terms. We first describe syntactification, and then how late-bound code generation makes use of this conceptual approach. Since our resulting translations are not opaque, and so may be inspected, they provide an intermediary for pragmatic control over the execution of a function. Instead of invoking a function directly, its translation may be first obtained, analysed, and optimised, prior to being explicitly issued for execution on either local or non-local hardware.

Syntactification is an approach to translation whereby translations are progressively refined from simpler less elaborate forms, to complex more elaborate forms. During the elaboration process, more information is made manifest in the translation, which was previously left abstract and implicit. Late-bound code generation is one particular method for performing syntactification, whereby individual elaboration steps are performed by modular code generators—the application of a particular code generator to the translation is selected by dynamic dispatch. Code generator modularity, and the use of dynamic dispatch for selecting the next refinement to the translation, are both defining characteristics of late-bound code generation.

2.2 Syntactification

Consider a programmer wishing to translate some portion of a program, perhaps even a running program, into a different language. The target language for this translation might be the machine language for a particular piece of hardware, or perhaps another high-level programming language. Furthermore, the translated program could be intended either for execution locally, or on some remote machine accessible over a network. How might they achieve this objective?

Semantically speaking, this problem has a trivial solution. Suppose the source program is written in Python, and the translation target is Java: first create a closure abstracting the portion of the program for which a translation is sought, for example:

```
def f():
    image = take_photograph()
    face = detect_face(image)
    if face.id in permitted_personnel:
        open_door()
```

One possible translation of `f` into Java is:

```
g.invoke();
```

provided that within the context of execution `g` has been bound to a *proxy* for `f`. This proxy might necessarily need to communicate over a network, to reach the process within which the closure instance for `f` resides. Invocation of the proxy `g` simply forwards the command to the original Python process, and executes `f` there. Here we have a rather general solution, since this singular Java translation equally serves not only for this definition of `f`, but for many other possible definitions!

However, such a translation might appear rather unsatisfactory. Perhaps the following fragment might better align with typical expectations for what an acceptable translation looks like:

```
Image image = take_photograph();
Face face = detect_face(image);
if (permitted_personnel.contains(face.id)) {
    open_door();
}
```

Supplementary Java translations for `detect_face` and perhaps also `permitted_personnel` might be supplied, compiled, linked, and executed within the target process. This is clearly a possibility if `detect_face` is a *pure* function, since the entire behaviour of the function can be simply reformulated with respect to Java primitives. Similarly if `permitted_personnel` is a *constant* list its contents may be serialised and attached to our translation.

In contrast however, `take_photograph` and `open_door` pose a more significant challenge. In this example both require access to a physical resources, a camera and an electronically controlled door, respectively—there are not necessarily any equivalent cameras or doors against which to rebind, without altering the system's semantics. Our trivial proxy based solution may be deployed again, but here to just a subset of the program: both `take_photograph` and `open_door` must be bound to proxy objects in the Java process, which delegate to the pair of identically named functions in the original Python process.

Other, more sophisticated solutions also exist. For example the behaviour of `take_photograph` and `open_door` might be split into parts, depending and not depending on the singular resource. That part not depending on the resource could still be included in the translation. Another possibility is that another route directly from the Java process to the resource could exist, and be substituted to remove any indirection through the Python process.

Furthermore, if instead `permitted_personnel` were *mutable*, and we insist on our translation remaining semantically sound, this object would need to be proxied also, leaving only the invocation of `detect_face`, the attribute access of `face.id`, and the conditional `if` operation to execute within the Java process.

Although our second more satisfactory translation has not removed *all* need for proxy objects, it has arguably reduced their use to some kind of minimum. In many other, common scenarios the minimum number of proxies required often happens to equal precisely zero. But, even if only `detect_face` can execute within the target process, this could offer material benefits: the face detection algorithm might be implemented using a neural net, for which the machine hosting the target process could have specialised hardware acceleration, not available to the Python process.

Why then should we be more satisfied with the latter translation, when it is semantically equivalent to our trivial solution, and similarly relies on proxy objects? As we alluded to above, it has to do with the extent to which operations are executed within the process executing the translated program, as opposed to simply delegated for execution within the original process. A necessary requirement for an operation to execute within a particular process is to encode that operation syntactically within that process state—after all the physical processor must eventually consume a syntactical representation of the operation in its native machine language. Briefly put, this latter translation is more *syntactically elaborate*.

From this perspective, the aspect of translation which is non-trivial is the process of discovering binding and behavioural information about bound objects, and representing that syntactically—typically inline within the translation. In short: transforming information from scope to syntax. This process we call *syntactification*.

Syntactification may be performed progressively, as a kind of reduction. For instance, we might start with our initial translation, and perform a series of discrete syntactification steps in order to arrive at our final translation:

```
(1) g.invoke();
```

```
(2) c = x.invoke();
```

```
    if (c) {
        open_door();
    }


(3) Face face = y.invoke();
    if (permitted_personnel.contains(face.id)) {
        open_door();
    }


(4) Image image = take_photograph();
    Face face = detect_face(image);
    if (permitted_personnel.contains(face.id)) {
        open_door();
    }
```

Each progressive translation is equally valid, yet improves on the former in the sense that it is syntactically more explicit.

Adopting this reductive perspective on translation is not simply an esoteric thought experiment, but a perfectly reasonable basis from which to build a compiler. The translation system introduced herein is founded upon this very concept of *progressive syntactification*. Each translation problem is framed with respect to its *default translation*—corresponding to the trivial proxy-based solution above—and improved upon by a set of modular code generators. The full system that we describe also supports proxies that take arguments.

One notable consequence of this approach is that the intermediate representation language for the translation is the same as the target language itself, although translations may be performed explicitly to target any synthetic language, constructed to act as a convenient intermediary if desired. During the course of a translation, many proxies which are introduced during early syntactification steps are subsequently found to be unnecessary, upon completion of later syntactification steps. In certain situations all proxies can be eliminated, leading to a *separable* translation.

Precisely how a particular code generator is selected to perform the next syntactification step is a question of design, since many serviceable possibilities exist. In the system we present, code generators are selected by means of dynamic dispatch, with the ability for more specialised code generators to dynamically override less specialised code generators, as advocated within the design philosophy of object-oriented languages and systems. This novel, dynamically dispatched approach to syntactification we term: *late-bound code generation.*

* * *

The task of a late-bound code generator is to discern both binding and behavioural properties of the proxies it is applied to, and generate a faithful representation of that behaviour in the syntax of the target language. This is typically accomplished by further delegation, and indirect mutual recursion between the set of code generators. From the perspective of the code generator, we often refer to the proxy objects they act upon not as proxies, but more generally as *providers.*

Late-bound code generators are typically associated with a particular source function, or limited class of functions, and also with the target language for which they are capable of generating code. This allows highly specialised code generators to be supplied, in order to generate optimised code for that specific function and target language combination—often implementing some kind of domain specific optimisation. Completely generic code generators are occasionally useful, however as we shall explore in Chapter 5, seemingly generic code generators, such as those for the source language's primitive operators, may typically be associated with an appropriate specific procedure, for instance methods of a metacircular evaluator for the source language.

What gives late-bound code generation its characteristic flavour is that the object under manipulation during translation can be thought of as a *syntactic closure* [BR88], expressed in the target language, but closing over the source process. It is this very property which allows for a semantically sound, yet trivial, translation by default, and for the syntactification process to safely halt after any step, without invalidating the translation.

Given the modular nature afforded by this dynamically dispatched design, code generators must all conform to a common interface. In the following sections we briefly outline the form of this unified interface, and the operator used to pose translation problems to our system.

2.3 Introducing `translate`

We introduce `translate`, our novel operator for performing late-bound code generation: `translate` acts on function *instances* taking operands dictating the target language and also information related to any formal arguments, and returns a syntactic closure in the given target. This returned fragment implements the *invocation* behaviour of the source function, in the target language.

The ubiquitous `invoke` operator—available for calling a function in almost all contemporary programming languages—usually offers the following familiar interface:

```
⊤ invoke(Function function, List<⊤> arguments)
```

where ⊤ denotes the universal type (refinements of this interface are of course possible). It is usually applied not as an function, since that would lead to infinite regress denotationally, but using syntax built into the language grammar. In contrast the interface of the `translate` operator is defined as follows:

```
Target translate(
  type Target,
  Target function_provider,
  List<Target> argument_providers,
  ⊤ mandate
)
```

Whereas the `invoke` operator takes a function instance and a list of argument values, performs an allowably side-effectful computation with those arguments, and returns a single value, the `translate` operator does not issue any of the function's side-effects when applied. Instead it is an operator for the compilation of a function's behaviour to the given language: `Target`. The value returned by `translate` is a term in the target language representing all computation entailed upon an invocation of the *provided* function—precisely how the function is provided is explained next. Note that `translate` requires the target language to be identified via its type (or class) such that instances of type `Target` correspond to terms in the target language.

Providers

To directly contrast `translate` with `invoke`: whilst `invoke` takes a list of argument values, `translate` takes instead a list of argument *providers*. An argument provider represents how the argument value is to be delivered to the translated term—we note that the providers supplied to `translate` are always terms in the target language. For example, when targeting a register machine an argument provider might be a single instruction to load a register with a value from memory. When targeting a hardware description language, an argument provider might instead be the output buffer of a multiplexer. Since a provider is a full term in the target language, the argument might even be computationally derived within the provider. If an argument is required by the computation, its provider will usually be incorporated (at least once) as a sub-term of the term returned by `translate`.

Given that `translate` is an operator acting on functions, it might appear curious that the function itself is not a direct operand to `translate`. This is a necessary consequence of its role as an operator for compilation. Within a compilation context there is not always enough information available to fully instantiate every function being compiled. For example it is quite common that the syntax of the function's implementation is known, but its lexical scope is immaterial (perhaps only some partial template is known, or some concrete ancestor). Hence, in general the function is supplied to the `translate` operator as a term in the target language, in exactly the same way as its arguments: specifically a term which within the context of the translation provides the function instance.

Fortunately, in many common cases the function is already instantiated at the point of compilation, in which case it may be supplied as an *immediate provider*, and accessed directly during translation as one might expect. This naturally raises the question of how to deal with situations when some amount of useful information is known about the function, but not fully what is required to supply an immediate provider—we delay this important discussion to later sections.

Mandate

The fourth and final operand to `translate` is an optional value which we refer to as the *mandate*. The mandate offers an opportunity to transfer information about the dynamic compilation context to the underlying code generator. The mandate is a *compile-time dynamic variable*, and is usually superfluous—however in certain situations it is indispensable, as we shall discover later.

2.3.1 An example

To build an intuition for the behaviour and utility of the `translate` operator, consider the following short Python program:

```python
def fahrenheit(celsius):
  print(celsius * 9 / 5 + 32)

fahrenheit(22.5)

mandate = { 'python_builtins_binding': 'const', 'shared_stdout': True }

translation = translate(
  Scheme,
  immediate(Scheme, fahrenheit, mandate),
  [ Scheme('input') ],
  mandate
)
```

Running this program causes the following to be printed to the console:

```
  72.5
```

As expected, invocation of function `fahrenheit` first causes the computation converting the argument in Celsius to Fahrenheit to be issued to the underlying hardware platform—exactly how is implicit. Then the result of the calculation is printed to the console, using the side-effecting function `print`.

Applying `translate` to the function `fahrenheit` on the other hand has quite different behaviour, in particular nothing is printed to the console. Instead a term representing the behaviour of `fahrenheit` under invocation is generated, in the requested language, and returned to the caller. Consequently, before the program exits, a representation of the term in the Scheme programming language

```
        ( let ( ( v0 ( + ( * input ( / 9 5 ) ) 32 ) ) )
                ( begin
                        ( display v0 )
                        ( newline )
                )
        )
```

is bound to the global variable `translation`. We will discuss exactly how this term is generated in later sections. For comparison, the default translation, before syntactification is as follows:

```
( ( origin-proxy #x7f7ddcbf2fc0 "ctypes.py_object" ) input )
```

Here origin-proxy simply constructs a proxy for the Python object fahrenheit in the originating interpreter, located at virtual address 0x7f7ddcbf2fc0. This translation, whilst trivial, requires the connection between both interpreter processes to remain active, in order for the translation to be executable. It is also required that a valid implementation of origin-proxy is present. In contrast, the *eventual* result of the translation is *separable*: enough information has been syntactically elaborated, such that the translation is executable without any form of interaction with the originating process.

Once represented as a data structure, this Scheme term representing the behaviour of fahrenheit may be computed upon arbitrarily. It might be reorganised, issued to some local coprocessor selected by the program, or even distributed for execution on an array of remote machines. We have thus attained pragmatic control over the execution of fahrenheit. For example, we might issue the translation for execution under a lower priority subprocess—which would be considerably more useful if fahrenheit entailed a long-running computation:

```
import subprocess
subprocess.run(['nice', '-n', '10', 'guile', '-c',
    '( let ( ( input 22.5 ) ) ' + str(translation) + ' )'
])
```

It is worth noting that this data structure, representing the Scheme term, was generated without any need to modify the interface of fahrenheit—for example, it was not passed as the function's return value, as would be required in MetaOCaml or LMS. Neither was it required to rewrite fahrenheit in a multi-level language—whereby the function definition includes names bound only at some future stage, and only operations which depend on them are delayed in their effects.

The operands in detail

The four operands of our example warrant further explanation. The first operand Scheme is the type, or—equivalently in this example—class, representing the target language of the translation. The value returned by `translate` is guaranteed to be an instance of `Scheme`, representing a term in the Scheme programming language. This resulting term should faithfully implement the behaviour of `fahrenheit` under invocation. The target `Scheme` has been designed such that *evaluation* of the translated term (according to the semantics of the Scheme programming language) performs an equivalent computation to the *invocation* of the provided Python function.

The second operand is supplied with the assistance of the function `immediate`, which simply generates a term to proxy the given object in the target language. These proxies serve the purpose which we outlined in Section 2.2, and are often elided from the eventual translation, as in our example above.

The third operand is a list of argument providers. An argument provider describes how the argument value will be delivered to the translated term. In this example we have arbitrarily decided to provide the one and only argument to the translated term as a lexical binding of the Scheme symbol `input`. The contract we are establishing with `translate` is that the translated term will be, and will only be, evaluated within a lexical scope in which the symbol `input` is bound. Furthermore we are guaranteeing that the binding resolves to a Scheme number, a number representing the argument in degrees Celsius. This provider may be used zero or more times within the translated term, but is strictly valid only in the context which directly interfaces with the translated term—the translated term may shadow the binding and so must arrange to evaluate any providers exterior to the shadowing scope. The question of hygiene in translation is a very interesting one, which we shall return to again in future sections. By the term “hygiene” we mean ensuring that name bindings never resolve to an *unintended* value: splicing a code fragment which assigns to a temporary variable via a name shared with the surrounding code is an example of hygiene violation, since subsequent surrounding uses of that name would be rebound to the unintended temporary value. Hygiene may be maintained either manually by the programmer, or automatically within the language system.

For purposes of demonstration we chose to name the Scheme symbol `input` rather than `celsius` to hint at the manner in which `translate` preserves hygiene. Names for bindings are never automatically transferred from source terms to translated terms, which beyond being a potential hazard to hygiene would fail to be well defined for certain target languages. Instead bindings in the target language are always explicitly supplied. However, supplying names in an undisciplined manner is also a hygiene violation in the waiting—throughout the remainder of this thesis we will use an automatic name generator, which guarantees full binding hygiene.

The fourth and final operand supplies the mandate of the translation. In this example the mandate is a simple dictionary, which asserts that (i) names resolving to Python's builtin functions should be treated as constant references to those functions, and that (ii) the translation may assume that the translated term will be evaluated in a process sharing the "standard output" file of the Python interpreter—this is indeed the case in our example of a Scheme interpreter forking the Python process. Together these statements allow the invocation of the Python function `print` to be translated into applications of the Scheme functions `display` and `newline`. Evaluating this translated term in a process not sharing a standard output file would not be equivalent to invoking the original `fahrenheit` function, since the converted temperature would be written to a different file or console.

We did not choose to enable optimisations within our example mandate. Had the mandate instead enabled optimisations:

```
mandate = {
  'optimise': True,
  'python_builtins_binding': 'const',
  'shared_stdout': True
}
```

then the division of constants 9 and 5 could have been folded to produce the translation:

```
( let ( ( v0 ( + ( * input 1.8 ) 32 ) ) )
  ...
)
```

where the dividend and divisor have been constant folded at translation time.

Having seen a simple example of applying the `translate` operator, a few questions surely come to mind. The first of which is likely: how exactly is the translation performed? So far we have neglected to mention any detail of the `translate` operator's implementation, and focused exclusively on its application and interface. Schematically, the implementation of `translate` is our topic for the following section.

2.3.2 Implementing `translate`

The most basic implementation of `translate` is rather trivial: the applied operands are simply dispatched to a *code generator* associated with the provided function (or another appropriate abstraction in the case of non-immediate functions, as we shall see later). Consider again our example:

```
translation = translate(
  Scheme,
  immediate(Scheme, fahrenheit, mandate),
  [ Scheme('input') ],
  mandate
)
```

Let us imagine that our language implementation, for example the Python interpreter, maintains a table relating function instances with programmer-supplied code generators:

Function	**Target**	**Code Generator**
`fahrenheit`	`Scheme`	`translate_fahrenheit_to_scheme`
`print`	`Scheme`	`translate_print_to_scheme`
`fahrenheit`	`Verilog`	`translate_fahrenheit_to_verilog`
...	...	...

These code generators each target one particular language: for instance some code generator named `translate_fahrenheit_to_scheme` might be responsible for generating a representation of `fahrenheit` in Scheme. Note that despite appearances we are not implying a textual association here, but instead a table associating function instances and target types with code generator instances.

Given such a table, the `translate` operator may be implemented to simply delegate to the relevant code generator, given the required target language. It is then the responsibility of the code generator to perform the requisite translation. The job of the translator is simplified however, since its task is specialised to one function and one target language.

This requires that the programmer writes, or at least installs, a code generator for each function to be translated. Therefore such a simple, table-based implementation of `translate` is not particularly useful, but it serves as a basic model for understanding code generators in their simplest form—indirection via such a runtime table serves as a minimal mechanism for our dynamically dispatched operator. We shall return later to refine this basic mechanism in Chapter 5, including extensions for performing translation by means of reification.

Assuming our table-based design, `translate` might be implemented in Python as follows:

```python
def translate(Target, function_provider, argument_providers, mandate):
  assert function_provider.is_immediate() # Assume the function is immediate
  function = function_provider.immediate_reference()
  code_generator = translation_table.lookup(Target, function)
  translation = code_generator(
    function,
    argument_providers,
    mandate
  )
  return translation
```

Notice that here we are suggesting how translate might be simulated within a Python program, rather than how the Python language might be extended to include the operator natively. In many cases it is possible to simulate `translate` in this manner, but it is not true in general, specifically in those cases where the host language does not provide sufficient introspective and reflective facilities for a full simulation: this is why we generally refer to `translate` as an operator, rather than a function.

Clearly the interface presented by the code generator is similar to that of `translate`. In our example the interface to the code generator translating `fahrenheit` to `Scheme` is as follows:

```
  Scheme translate_fahrenheit_to_scheme(
    Function function,
    List<Scheme> argument_providers,
    ⊤ mandate
  )
```

The most important question therefore is not how `translate` is implemented, but how the code generators it dispatches to are implemented.

Code generators

Returning to our example, how might the underlying code generator be implemented? Since this code generator is specialised to the semantics of the function `fahrenheit` it can be expressed quite simply:

```
def translate_fahrenheit_to_scheme(_, argument_providers, mandate):
  celsius_provider = argument_providers[0]
  proform = generate_unique_symbol(mandate)
  print_term = translate(
    Scheme, immediate(Scheme, print, mandate), [ proform ], mandate
  )
  return Scheme.Let(
    [ ( proform, Scheme.Add(
      Scheme.Multiply(celsius_provider, Scheme('( / 9 5 )')),
      Scheme('32')
    ) ) ],
    print_term
  )
```

The translated term is constructed imperatively by concatenating each sub-term of the translation. The first thing to notice about this code generator is that is delegates some part of its translation, via a recursive application of `translate`. This is idiomatic of the style we adopt throughout when programming with `translate`: dynamically dispatched mutual recursion between code generators.

Specifically, our code generator for `fahrenheit` does not directly generate the terms for `begin`, `display` and `newline` which appear in that translation. These are generated by the code generator for `print`, to which the application of `translate` dispatches. Instead only the Scheme terms for the temperature conversion are generated here. The major challenges which arise with this approach, in which the task of translation is divided into sub-tasks each dynamically dispatched to their own code generator, are how to permit sufficient communication between sub-term generators, and how to compose the translated sub-terms correctly dealing with side-effects.

Side-effects & let-insertion

The possibility of unbounded side-effects requires that the evaluation order of translated sub-terms mirrors precisely the evaluation order of the source sub-terms. However, if bounds can be placed on the range of side-effects that may be issued by the function, then this rule can potentially be relaxed, or even elided. When writing code generators for Scheme we generally preserve the requisite ordering, as we have done in this example, by means of *let-insertion* [Rom16].

Let-insertion is a technique for systematically composing two terms into a single term, within which the side-effects of the two sub-terms are strictly ordered under evaluation, and whereby the second sub-term can depend on the result of evaluating the first. The preceding sub-term appears at the site of a `let` binding's value expression, and the succeeding sub-term within the body of the `let` expression. Within the translation, the result of evaluating the first sub-term is bound to an intermediate term which we refer to as the *proform*. The proform is a special case of a provider, used within the second translated sub-term to represent the result of evaluating the first sub-term.

Generating code strictly according to this pattern of let-insertion results in translations in a simple variant of administrative normal form (ANF) [Rom16]. Binding hygiene is maintained by ensuring that each generated proform is unique within the evaluation context of the translated term. The extent of this context may be customised by the mandate: for instance if the result of the translation is intended for composition with terms generated by an other independent Python processes, the mandate might ensure uniqueness by prefixing each proform with some string unique to each process. By systematically generating the name of every binding within the translated term using `generate_unique_symbol`, and never otherwise inventing or reusing names from the source term, hygiene-preserving code generators may be written straightforwardly.

Clearly in our example all of the arithmetic operations are side-effect free, and thus let-insertion was not strictly required. We wrote our code-generator conservatively, using let-insertion, to demonstrate the general case where side-effects are unbound. Given this observation, `translate_fahrenheit_to_scheme` may be simplified to:

```
def translate_fahrenheit_to_scheme(_, argument_providers, mandate):
  celsius_provider = argument_providers[0]
  return translate(
    Scheme, immediate(Scheme, print, mandate),
    [ Scheme.Add(
      Scheme.Multiply(celsius_provider, Scheme('( / 9 5 )')),
      Scheme('32')
    ) ],
    mandate
  )
```

whereby we have supplied the temperature conversion calculation within the argument provider to `print`, which according to the semantics of `translate` may appear *zero* or more times within the translated term. With this generator the resulting translation will be free of the unnecessary let-insertion:

```
( begin ( display ( + ( * input ( / 9 5 ) ) 32 ) ) ( newline ) ).
```

Generator registration

We have yet to address the question of *how* code generators are declared to the `translate` operator. There is substantial design freedom in implementing this mechanism. Given our basic table-based implementation of `translate`, all code generators must be registered in one global table, the `translation_table`. In this case, registration is simply a matter of appending a row to the table:

```
  def register_code_generator(Target, function, code_generator):
    translation_table.append(Target, function, code_generator).
```

Registration of `translate_fahrenheit_to_scheme` is performed thus:

```
  register_code_generator(
    Scheme,
    fahrenheit,
    translate_fahrenheit_to_scheme
  )
```

Whilst this registration mechanism is perfectly sufficient for translations involving only immediate function providers, code generator registration is quite a bit more complex in general. Later we will return to this issue, describing a registration mechanism built upon dynamic dispatch rather than a global table.

Soundness of the translation

Since `translate` performs all translation by means of delegation to arbitrary code-generating functions, there are no guarantees whatsoever as to the soundness of the translation. In principle each code-generator might supply a soundness proof alongside every translated term, however this idea is not explored further within this thesis.

Further abstraction

For purposes of exposition, we applied the `translate` operator *directly* in our example of translating `fahrenheit` to Scheme. Whilst this is perfectly legitimate, we expect that in most cases, users of the `translate` operator would use it as a basis for building further abstractions for pragmatic control. For instance, applications requiring offload of computation to the cloud might define a function `offload_to_cloud(function, argument_0, ...)` implemented using `translate`, but which presents an interface more convenient to the application domain.

Alternatively an application requiring considerable domain-specific optimisation for graphics rendering might define a function `render(canvas, draw_function)`, which first translates `draw_function` with respect to those optimisations, before installing the optimised translation into the requisite rendering hardware. This latter use case is expanded upon extensively in Chapter 4.

2.4 Conclusion

In this chapter we introduced late-bound code generation, a dynamically dispatched approach to translation based upon syntactification. Translation by syntactification always proceeds from a trivial seed: whereby any function invocation is simply translated to an invocation of a proxy in the target fragment. Technically this is achieved by representing the result of the translation as a syntactic closure, with its closing scope establishing the link back from the proxy to the proxied function. The translation is progressively elaborated, transforming (constant) behavioural and structural information about objects bound in the syntactic closure scope into terms of concrete or abstract syntax. The elaboration may safely halt at any point, and in those particular cases where all proxies are eliminated from the translation, this is known as a separable translation. Within syntactification, separable translations are a special case, and correspond to the traditional, purely syntactic notion of a translation.

We introduced late-bound code generation as a technique for performing syntactification, whereby individual elaboration steps are performed by dynamically dispatched code generator functions, associated directly with their proxied function (or in general, some class of proxied functions). Typically these code generators delegate particular sub-translations to other code generators: translation therefore proceeds by mutual recursion within the set of applicable code generators. To achieve this effect, code generators must all share a common interface, which we defined.

The principal operator of late-bound code generation, `translate` acts on (providers of) function instances, returning a syntactic closure in the target language, which implements the invocation behaviour of the provided function. The `translate` operator acts simply as a dispatcher, delegating translations to code generators according to its prioritisation policy. Invocations of the provided function may issue side-effects, the ordering of which is preserved in the translated fragment, typically by restricting target fragments to administrative normal form. We described the operands of `translate`, specifically the target language, the function and argument providers, which are syntactic closures, and the mandate, which pertains to the translation unit, for instance defining the optimisation level. We outlined how `translate` might be implemented, although discussion of the details is delayed until Chapter 5.

Chapter 3

Background and related work

3.1 Survey scope

Since late-bound code generation is a technique for programs to affect pragmatic control over their own execution, we limit the scope of related work surveyed herein to intra-program techniques for pragmatic *self-control* of program execution. By the term "pragmatic self-control" we mean that a program is exerting pragmatic control over its *own* execution—see Appendix A for a glossary of related terms. Specifically that excludes all techniques whereby pragmatic statements about the program's execution are asserted externally from the program text. Of primary interest are intra-program pragmatic statements which assert: where a particular computation must occur spatially; when a particular computation may occur in temporal relation to another; and which optimisations should be applied to a particular fragment of computation.

In contrast to intra-program techniques, extra-program techniques for pragmatic control are generally more familiar and widely adopted. Common extra-program techniques include compiler flags, manual deployment using remote access tools such as `ssh`, and more recently orchestration tools, notably Kubernetes [Ber14]. These tools typically operate upon program *files*—either source code files or object code files.

Directives, also known as "pragmas", are perhaps the most familiar intra-program technique known to professional programmers. A directive might apply to a single statement, a loop block, a function definition, or even an entire translation unit. The subject of a directive is typically dictated according to some lexical relation to the statement, and they are often used to control certain kinds of optimisation, such as whether or not a loop is unrolled or a function is inlined. Directives are typically informational, and convey advisory preferences for how a particular program fragment should be executed, to some meta-system. This meta-system ultimately affects the pragmatic control advised by the program.

In principle, however, any program which is capable of performing general input and output operations has the capacity to affect pragmatic control over itself—the program may explicitly issue commands to local or non-local hardware to perform any subset of its own computation. Explicitly issuing its own operations naturally affords a program completely general pragmatic control, as opposed to the limited range of control offered by directives, since it may fully customise the schedule and optimisations applied to the computation it issues. Although this ability is latent in all programming languages, it is not typically used, and in most applications almost all computation is implicitly issued for execution on the hardware into which the object program was loaded. Scheduling is often delegated at least in part to an external compiler or interpreter. Some common exceptions to this trend include 3D graphics engines, which preferentially issue geometric and rasterisation operations explicitly to a specialised accelerator, and object-relational mapping (ORM) libraries, which typically issue data relational operations explicitly for execution within a database system.

We refer to this phenomenon as “self-issuance”: by which we mean when a program explicitly issues instructions to hardware for performing its own computation, rather than implicitly relying on the language system to issue hardware instructions on its behalf. Although it is unclear why explicit self-issuance of computation is not more widely adopted, it is worth noting that whilst issuing operations is relatively trivial, the task of collating and representing the operations to issue is substantially more challenging. In most applications it would be relatively straightforward to send some object code file for execution on a remote server, compared to the difficulty of constructing the object code file itself—at least when setting aside concerns related to security. Consequently, those tools and research projects which do focus on the self-issuance of computation pay considerable attention to the primary challenge of constructing the syntactical representation of the issued operation. Obtaining this explicit syntactical representation of a program's own behaviour is a problem of *reification*.

In practice, when programmers do seek to explicitly issue computation, but convenient solutions for reifying the program's behaviour are wanting, they often resort to programming within an embedded programming language, rather than expressing that part of the program natively. For example, a Java programmer without access to an ORM library might express their database queries as SQL terms embedded as character strings within their Java program, rather than as methods operating on Java objects. Written in this way, the meaning of the query can no longer be determined given Java's semantics and the relevant class definitions. In general we refer to this approach as *embedded heterogeneous* programming—in contrast to expressing a program within one language, namely *homogeneous* programming. This is analogous to either the deep or shallow embedding of a domain specific programming language [GW14]. Several embedded heterogeneous programming techniques exist, exhibiting various degrees of sophistication and safety. Unavoidably, when expressing the behaviour of a program in separate fragments, each in a different programming language, the compositionality of these fragments becomes a major concern.

To summarise, intra-program techniques for pragmatic self-control typically fall into one of three (informal) categories:

- Type I: Directive-based control;
- Type IIa: Self-issued computation from a reified lexically homogeneous program;
- Type IIb: Self-issued computation from a lexically embedded heterogeneous program.

Specifically we refer to the *lexical* homogeneity or heterogeneity of the program, since the runtime state of type IIa and type IIb programs typically both contain objects which denote terms in two or more languages.

In the following sections we take each category in turn, and illustrate its character by means of a number of examples. We emphasise here that these categories are *thematic*, and not formally delineated—they are intended only to emphasise certain pragmatic control patterns—and individual approaches or technologies can, and often do, span more than one category.

We intentionally do not include within our survey technologies for macros, or domain-specific languages. Macro technologies have typically focused on compile-time semantic abstraction, whereas within this thesis we are primarily focused on run-time methods for pragmatic control. To the extent that macros have been employed for pragmatic control—as opposed to semantic abstraction—we believe that multi-stage programming languages present a comparable and more general run-time technique for code generation. Technologies for domain-specific languages are again typically focused on programming abstraction, although they might support directives, which we survey in Section 3.2, or they might be implemented on top of principled technologies for pragmatic control, such as multi-stage programming languages, surveyed in Section 3.3.2 and Section 3.3.3.

3.2 Type I: Directive-based control

One common method for affecting pragmatic control over the execution of a program involves the application of directives within the program text. Directives typically take the form of statements—otherwise known as pragmas—which annotate regions of the program with pragmatic requirements or preferences, and are orthogonal to the program's semantics. There are various kinds of directive, notably including those for controlling parallelism and optimisation.

3.2.1 Directives for controlling parallelism

One notable class of directives for controlling parallelism are those defined within the OpenMP standard. Originally specified for the Fortran programming language, and subsequently extended to include C and C++, OpenMP is an application programming interface (API) for writing shared-memory parallel programs for execution within a multiprocessing context [DM98].

OpenMP gives programmers control over the extent to which sections of their program execute in parallel, the degree to which data is shared between concurrent threads of execution, and primitives for synchronising control and data between those threads.

When using OpenMP the programmer typically defines a directive, lexically preceding the fragment to which it applies, specifying some pragmatic requirement for the execution of that fragment. For example, the following C program contains an OpenMP directive specifying that iterations of the `for` loop should execute in parallel.

```c
#include <math.h>
int main(void) {
    float results[256];
    #pragma omp parallel for
    for (int index = 0; index < 256; index++) {
        results[index] = sinf(index * M_PI / 128);
    }
    return sum(results);
}
```

Executing this program causes some number of threads, which may in general be determined at runtime, to be scheduled in parallel, each executing the loop body for some range of indices. Various supplementary clauses may be supplied, to specify more precisely aspects of the parallelism—such as the scheduling policy, or how shared variables are combined when threads eventually join.

Furthermore OpenMP provides a range of other kinds of directive, including those for controlling the atomicity of operations as observed by other threads, duplication and visibility of shared state, and also synchronisation barriers.

Several broadly similar APIs to OpenMP also exist, making use of pragmas to affect pragmatic control over C programs, but which instead focus on offloading computation to heterogeneous hardware accelerators such as GPUs and FPGAs: most notably OpenACC [NCPC11] and HMPP [DBB07].

3.2.2 Directives for applying optimisations

A particularly interesting class of directives for controlling the application of compiler optimisations can be found within high-level synthesis (HLS) tools. A diverse array of HLS technologies have been developed historically—within both academia and industry—a number of which have employed directives for controlling the exploration of the combinatorially large design space [CGMT09, MS09].

Xilinx, a contemporary leading vendor of field-programmable gate array (FPGA) devices, provide HLS tools for their FPGA products. Their compiler suite Vitis HLS accepts a number of directives for controlling the synthesis process [Xil21]. In addition to typical directives for enabling and disabling the inlining of functions, many of Xilinx's directives are specifically tailored towards exploiting the particular advantages of FPGA devices. For instance, the following pragma specifies a target latency for loop iterations:

```
float reduce(float values[256]) {
    float result = 0.0f;
    #pragma HLS latency min=10 max=20
    for (int index = 0; index < 256; index++) {
        result += values[index] * project(index * M_PI / 128);
    }
    return result;
}
```

If the compiler is able to synthesise an implementation of this loop body with less than 10 clock cycles of latency, then it will attempt to trade a reduction in resource utilisation for a higher latency. For example, sharing the same floating-point multiplication unit for both multiplication operations would reduce the physical footprint of the implementation, at the expense of increased latency, when compared to an implementation using duplicated multiplication units. Alternatively, moving computation from digital signal processing (DSP) blocks, which some FPGAs provide for accelerating numerical operations, to non-DSP blocks would reduce resource utilisation of one kind of scarce resource for an increase in another less scarce resource, again at the expense of increased latency.

On the other hand, if the synthesis of the loop body initially exceeds 20 clock cycles, the compiler will attempt to iterate the implementation in order to achieve lower-latency. In principle this might be achieved by selecting lower latency functional units for the implementation, or by adjusting placement decisions to mitigate the need for buffers. Typically latency can be reduced by duplicating resources, reversing the trade-off described above for increasing latency—in this example the latency of the implementation could be reduced with an implementation comprising two multiplication units, compared to one.

Another of Vitis HLS's notable directives is the `stream` pragma. Functions that take arrays as parameters are by default synthesised using a random-access memory (RAM) interface, with dedicated slots allocated for each element of the array. It is however quite common for functions consuming an array to access the array in a structured manner, typically to read each element only once and in order. In these cases, and additionally when the elements of the array are written sequentially, the `stream` pragma may be applied. The consequence of applying this directive is to eliminate the RAM storage at the function interface, and replace it with a shorter FIFO queue, through which the values of the array are delivered to the function in a streaming fashion.

Other powerful directives are also available, including the `pipeline` and `dataflow` pragmas, which offer control over the pipelined parallel execution of loop iterations and function invocation, subject to loop carried and dataflow dependencies.

3.2.3 Directives for defining optimisations

Beyond simply controlling the application of optimisations, directives can also be used for defining novel optimisations. For example, Haskell provides a sophisticated directive, called `RULES`, for specifying rewrite-based optimisations to the compiler [PJTH01]. Using Haskell's `RULES` pragma, programmers may specify application-specific identities over terms, which might be profitably applied by the compiler. This allows the programmer to extend the compiler with certain classes of domain-specific optimisation. The body of the rule consists of an equation, asserting an identity without proof—the compiler takes the validity of the identity as implicit. The rules language includes syntax for universally quantifying variables that appear within the rule, allowing for some degree of control over how bindings within the rule are resolved. By convention, the right-hand side of the rule's equation is considered preferable, and the compiler will attempt to rewrite instances of the left-hand side to this preferred form.

Although there are certain constraints on the form of legal rules, it is expressive enough to define a range of powerful and practical optimisations: for instance, list-fusion optimisations to eliminate the allocation of intermediate lists in functional applications, and specialisation operations to replace applications of generic functions with applications of faster type-specialised versions [PJTH01].

Besides the limitations inherent to rewrite-based optimisations, whereby programmatic analysis is limited to defining rules ahead-of-time, there are important practical issues encountered by Haskell's rewrite system. The foremost of which is rewrite *ordering*. Various competing, elaborate systems exist for programmatically defining rewriting strategies, notably Visser's System S [Vis99], such that the precise ordering of rule application can be controlled. In contrast, Haskell adopts a simple phase-based solution, allowing programmers to annotate rules with a phase-control specification, typically encoded as a number. This number dictates within which predefined internal phase of the Haskell compiler the rule should be applied. Such phase ordering is considerably less precise when compared to System S, which limits the range of optimisations that can be practically expressed. Furthermore, since phase-control numbers directly relate to the internal architecture of the compiler, the practical efficacy of a rule is not necessarily stable between different compiler implementations.

3.2.4 Directives for language subsetting

One less common application of directives involves language subsetting, whereby the directive proclaims that the fragment to which it applies is restricted to a particular named subset of the programming language. This restriction might be purely syntactic in nature, purely semantic, or both. When combining syntactic and semantic restrictions it can occasionally be arranged that the joint restriction can be verified only via syntax analysis: an example of this can be found in JavaScript, and its `"use asm"` directive [HWZ14].

The primitive datatype operations of the JavaScript programming language have in general relatively complicated semantics compared to those of C or Java. For instance, all numerical operations have floating-point semantics, and collections can in general hold values of mixed types, and both grow and shrink dynamically. Despite the fact that any given program or function might not exercise all such aspects of the language's semantics, it is left to the compiler to determine whether or not operations can be specialised. This task might rely on static analysis, information gathered by dynamic profiling, or both. Since the compilation cost of this analysis must be considered and limited, there are necessarily situations in which optimisation could legitimately be performed, but is not.

To help guide JavaScript compilers toward profitable and tractable opportunities for operation specialisation, a peculiar directive was proposed and widely implemented: `"use asm"`. Functions prefixed by this directive notify the compiler that the following function definition conforms to a *strict subset* of the JavaScript language known as *asm.js*. This language subset is restrictive enough to make type inference statically tractable, including sub-types such as the 32-bit integral sub-type implicit within JavaScript's floating-point numeric type. Upon encountering a `"use asm"` directive, the compiler first validates that the function definition conforms to the asm.js language subset. If this validation passes then the compiler can be certain that the application of various kinds of aggressive optimisation to this function definition, which could not be practically applied in all contexts, are valid and are likely to be profitable.

The following illustrative function definition conforms to the asm.js language subset, announced by the "use asm" directive:

```
function sum(x, y) {
    "use asm";
    x = x | 0;
    y = y | 0;
    return (x + y) | 0;
}
```

Note the presence of bitwise-or operations, coercing all values to 32-bit integers, according to JavaScript's semantics. This coercion allows the compiler to statically infer the type of x, y and the return value as a 32-bit integer, and therefore implement the + operation simply using machine native integer addition (assuming this is available).

Writing code directly in asm.js is typically rather cumbersome, and it was primarily envisioned as an intermediate target language for compilation, rather than a practical language subset directly for use by programmers. Most other languages enable the kinds of optimisation intended by asm.js through a combination of literal type declarations and compiler flags, rather than directives. However the general approach of selectively subsetting a programming language, via the application of directives, is a noteworthy and powerful mechanism for guiding the efforts of an optimising compiler.

3.3 Type IIa: Self-issued computation from a reified lexically homogeneous program

3.3.1 Syntactic quasiquotation

Plain syntactical quotation, and its later extension to quasiquotation, was first developed within the context of the Lisp programming language [Baw99]. The key idea is to introduce an operator into the language, applicable to literal terms within the program's definition, which suppresses evaluation of the applied term and instead returns the syntax of that term as a data structure. In principle the returned structure might be explicitly issued for execution on local or non-local hardware, perhaps after translation, optimisation or other forms of analysis. In practice however, quasiquotation often received considerable attention as a technique for defining macro transformations—with the goal of achieving abstraction—rather than primarily as a mechanism for pragmatic control [CW20].

Quasiquotation differs from plain quotation in that certain sub-terms within the quoted term may be annotated to escape from that quotation: the evaluation of those escaped sub-terms is not suppressed, and the result of their evaluation is then inlined into the quoted term, replacing the unevaluated sub-term.

One major perceived advantage of quotation and quasiquotation, over explicit syntax construction which we discuss later in Sections 3.4.1 and 3.4.2, is that the reified term is denoted in exactly the same manner as any other source term—although this fact naturally limits quotation to the type IIa homogeneous context.

When applying quotation or quasiquotation as a technique for pragmatic control, whereby analyses of the quoted term are required, as opposed to the simple syntactical transformations required by macros, one must address the question of how terms that are free within the quotation are bound. In general, other metaprogramming facilities, or static assumptions, could be called upon to resolve this issue. Alternatively, and perhaps more commonly, the quoted term is analysed without context, whereby all free sub-terms are left unbound.

3.3.2 Annotation-directed multi-stage programming

Multi-stage programming languages provide a mechanism for explicitly delaying code execution, known as *staging*. Staging a term prevents its computation from happening in the current evaluation context, instead providing the delayed term as a *code* value which can be either explicitly executed at some subsequent point in the program execution, or composed with additional code values to form more complex code values. Staging differs from both purely syntactic reification and functional abstraction, although it shares similarities with each. The programming language MetaML [TS00] and its more recent counterpart MetaOCaml [CTHL03] are both examples of annotation-directed multi-stage programming languages.

Syntactic reification, either plain quotation or quasiquotation, does not in and of itself preserve hygiene—see Appendix A for the definition of hygiene used within this thesis. Names used within the quoted term are detached from their lexical scope at the point of reification, and so risk becoming rebound to semantically *different* values or variables when the reified term is evaluated. Consequently the burden is placed on the programmer to ensure that hygiene is preserved (to at least the requisite extent) when employing quotation. Whilst multi-stage programming languages do typically provide an operator superficially similar to quotation, whereby the operator reifies the applied term, this staging operator enforces and automates the preservation of hygiene. Consequently, the reified term is not simply syntactic, since it preserves the lexical scope of any names free within the term, closing over its lexical scope similarly to a function closure.

Characterising multi-stage programming languages

From a semantic perspective, staging is strikingly similar to functional abstraction. The operator for functional abstraction (λ, `lambda` or in ML `fun`) also delays the computation of the applied term, providing a value for subsequent and explicit invocation, to affect the delayed computation. Furthermore λ also preserves hygiene by closing any free names over the lexical scope of the definition. Two significant differences between functional abstraction and staging are that: (i) unlike closures staged code values do not typically accept formal arguments on execution; (ii) multi-stage programming languages often include an operator for escaping sub-terms within a staged term. This second point requires further examination—consider the following MetaOCaml program:

```
let square z = z * z
let polynomial a b x y =
    (square a) * x + (square b) * y


let lift v = .<v>.
let specialised_polynomial_code a b =
    .< fun x y .~(lift (square a)) * x + .~(lift (square b)) * y >.


let polynomial_5_6_code = specialised_polynomial_code 5 6
let polynomial_5_6 = Runcode.run polynomial_5_6_code


let slower = polynomial 5 6 10 20
let faster = polynomial_5_6 10 20
```

First we define the function `polynomial` which computes the degree-one polynomial $a^2x + b^2y$, where a^2 and b^2 are considered coefficients of the polynomial, and x and y its indeterminates. Suppose this polynomial is to be evaluated several times, each against different values of x and y, but for the same values of a and b. Rather than computing the square of a and b repeatedly for each evaluation, it would be more efficient to precompute these squares once, ahead of time. To this end we define a function `specialised_polynomial_code` which returns a *code object* for defining a polynomial function, specialised to these constant coefficients. When we apply `specialised_polynomial_code` to `5` and `6` we obtain a code object equivalent to the term:

```
fun x y -> 25 * x + 36 * y
```

This is achieved by employing two of MetaOCaml's three staging operators: brackets and escape. MetaOCaml's staging brackets, written `.< ... >.`, delay the evaluation of the enclosed term and evaluate to a lexically closed code value, analogous to lambda abstraction. Inversely the escape operator written `.~ ...` may only appear within staging brackets, and escapes from the staged terms causing the applied term to evaluate within the context of the current stage, not within the delayed context of the staged term.

Enclosing our function definition within staging brackets delays the definition of our specialised polynomial function to some subsequent stage. However, application of the escape operator allows us to compute the values of a^2 and b^2 ahead of time, within the current stage. The need for the application of `lift` to the precomputed value is because the escape operator requires the applied term to evaluate to a code object, and not an integer as returned by `square z`—this code object is then hygienically inlined into the enclosing term. Consequently we must *lift* this object to the future stage, using `lift` which returns a code object representing the given value.

The third and final MetaOCaml operator, written occasionally in the literature as `.!`, but invoked as `Runcode.run` above, evaluates code objects using the MetaML or MetaOCaml interpreter or compiler.

Comparison with functional abstraction

Whilst staging brackets behave analogously to λ, there is no corresponding analogy for the escape operator. We could consider defining such an operator—for example consider the following program written in an imagined extension of Python:

```
def square(z):
    return z * z

def specialised_polynomial(a, b):
    return lambda x, y: (escape square(a)) * x + (escape square(b)) * y

polynomial_5_6 = specialised_polynomial(5, 6)
result = polynomial_5_6(10, 20)
```

Here our imagined `escape` operator allows expressions to be evaluated at the point the function is defined, and their resulting values implicitly incorporated into the function definition via anonymous bindings in the closure scope. Hence from this perspective the major innovation of multi-staged programming appears to be within the escape operator, rather than within the staging operator itself.

Notably, the effect of the escape operator can be simulated when using functional abstraction, by moving the evaluation of those terms outside of and before the function definition. Applied to our example, this could be achieved by simply using an alternative definition of `specialised_polynomial` thus:

```
def specialised_polynomial(a, b):
    square_a = square(a)
    square_b = square(b)
    return lambda x, y: square_a * x + square_b * y
```

Whilst this particular example might lead to the impression that multi-stage programming languages do not afford any additional facilities beyond those already inherent in function abstraction, this is demonstrably not the case. Our definition of `specialised_polynomial_code` above, although perfectly legitimate and legal MetaOCaml, does not represent the style of programming encouraged by multi-stage programming languages—a more canonical definition might instead be written:

```
let canonical_specialised_polynomial_code a b x y =
    let square_a = square a in
    let square_b = square b in
        .< square_a * .~x + square_b * .~y >.

let also_polynomial_5_6_code =
    .< fun v0 v1 -> .~(canonical_specialised_polynomial_code 5 6 .<v0>. .<v1>.) >.
```

In this case, the definition of the polynomial function is made with respect to all four arguments, rather than as a function that takes coefficients and returns a function of indeterminates, as before. Consequently if all of the staging annotations were to be removed from this definition, it would read naturally as an unstaged definition of our polynomial function. It is for this reason that multi-stage programming languages refer to their staging operators as annotations, not to suggest the idea that they are purely informational and may be optionally heeded (which is not the case) but instead to suggest the idea that if they were to be removed an equivalent unstaged definition may be obtained.

Consider how `canonical_specialised_polynomial_code` is invoked. Here we arrive at the most notable aspect of this example, which illustrates an important characteristic of multi-stage programming languages: observe that the binding structure of `v0` and `v1` has no faithful analog when taking our analogy with functional abstraction. Here `v0` and `v1` are formal arguments to a staged function definition, whose bindings are related to the names quoted within the escaped term due to their lexical scope. That the semantics of this binding form are well-defined distinguishes multi-stage programming languages from languages lacking such staging operators. Clearly any naïve attempt to refactor the escaped term outside of and before the function definition leads to `v0` and `v1` falling outside of their scope. In our second Python definition of `specialised_polynomial` above, we offered one particular solution to this problem, by binding these values using the formal arguments of the returned closure. Kameyama et al. have formalised a translation from staged code to typed closures [KKS08] which allows staged programs restricted to two stages to be translated automatically to an unstaged functional program—notably this translation preserves both typing and hygiene, and is also conjectured to preserve termination and evaluation order.

Restrictions, guarantees and safety properties

In MetaOCaml all staged code is opaque, and the semantics of the language forbid any introspection facility that could reveal the syntactical structure of code values. This is motivated by a concern that such capabilities would void important compiler optimisations, by weakening the theory of the language's semantics [TN03]. As a consequence, all staged code must be generated strictly via composition. Although this precludes many common code generation optimisations, such as peephole optimisation, it has been demonstrated for some applications that well optimised code can be formed under this restriction [KST04].

Furthermore, multi-stage programming languages enforce the *inlining* of terms which are incorporated into staged code using the escape operator. This is in contrast to closures, which typically do not, and in the presence of unbounded recursion cannot commit the compiler or interpreter to a full-inlining strategy. Therefore when composing terms in a multi-stage programming language, the programmer is assured that any overhead due to defining the staged code in a modular fashion, using several separate generic functions, is only incurred during the execution of the current stage, and factored out of the future stage.

Whilst modern JIT compilers such as the OpenJDK's HotSpot are capable of fully inlining deeply nested closures, for instance 15 or more levels deep, such aggressive inlining is not performed by default and it must be enabled using specific compiler flags. Even so, a JIT compiler will typically not guarantee that all overhead due to abstraction will be removed, and might decide to retain certain indirections within the optimised code if they reduce the footprint of the generated machine code—given the finite size of typical CPU instruction caches this decision might improve *both* execution speed and memory footprint, but the lack of a guarantee as to the compiler's decision gives rise to a source of performance non-determinism.

One particularly important aspect of MetaML and MetaOCaml lies within the type safety properties guaranteed by their compilers. A staged program which type-checks correctly at compile time is guaranteed to only *generate* code free of certain type errors at runtime. This is no trivial undertaking, and a substantial amount of theory has been established in order to provide such guarantees [TS00].

An interesting kind of type error peculiar to multi-staged programming languages is *scope extrusion* [KKS09, WRI+10]. Scope extrusion occurs whenever a variable which is defined in a subsequent stage is used in a preceding stage. In the following fragment of MetaOCaml the variable `x` is subject to scope extrusion, and thus the program is meaningless. Type systems capable of detecting all cases of scope extrusion at compile time have been developed, for example by Kameyama et al. in [KKS09].

```
let foo = .< fun x -> .~(bar x) >.
```

Finally, another interesting safety property relevant to multi-stage programming languages is the issue of *cross-stage persistence*. Cross-stage persistence is using a value from staged code which was defined or computed within an earlier stage of the program. In our definition of `canonical_specialised_polynomial_code` above, the values `square_a` and `square_b` must be persisted between stages for this reason. Since in MetaML and MetaOCaml the generated program must be executed within the same process as generated it, these values may be persisted on the process heap, similarly to a closure—if MetaML were to support code serialisation then the operational mechanism for cross-stage persistence would be a more complicated issue. Certain safety properties for cross-stage persistence are guaranteed by the compiler, in particular that: any value used in a subsequent stage will be persisted, and resolved according to a lexical scoping discipline; and also any value defined in a subsequent stage cannot be used from an earlier stage, an issue called "scope extrusion" [TS00].

3.3.3 Type-directed multi-stage programming

In contrast to annotation-directed multi-stage programming languages, Lightweight Modular Staging (LMS) is a *type-directed* approach to multi-stage programming [RO10], implemented as a library for the Scala programming language. Whilst Scala itself is not natively a multi-stage programming language, LMS extends Scala with support for multi-stage programming.

In many ways, LMS is similar to multi-stage programming languages such as MetaML and MetaOCaml, sharing their support for cross-stage persistence, lexical scoping of names within staged terms, compile-time type-safety guarantees, and enforcing the opacity of staged code from (most of) the program. However, LMS departs from traditional multi-stage programming languages in a number of important ways, most notably:

- Staged code is not denoted via syntactical staging annotations, but instead using *types*.
- The order of computation (including imperative control-flow) within blocks of staged code is automatically preserved—based on Scala's native evaluation order—rather than managed manually via syntactic composition.

- Access to the structure of staged code is permitted to certain components within the program, allowing for serialisation and computation offloading, as opposed to being opaque to the program entirely.
- The language in which generated code is represented need not match the source program, but may instead be represented in any language (for example in languages as diverse as JavaScript or SQL).
- Common subexpression elimination is performed automatically by the staging system.
- Optimisations can be declared in a modular fashion, building on Scala's powerful traits system, and included or composed at the discretion of the source program author.

Whilst most of these departures are fairly self explanatory, a number of these points require further elaboration.

Staging using the `Rep[T]` type

As we described in section 3.3.2, MetaML-like languages denote at which stage fragments of the program are executed using syntactical staging annotations, namely using staging brackets and the escape operator. In contrast staging code in LMS is directed by the type system—consider the following Scala definition:

```
trait Polynomial { this: Arith =>
  def polynomial(a: Int, b: Int, x: Rep[Int], y: Rep[Int]): Rep[Int] =
    (a * a) * x + (b * b) * y
}
```

The body of this function definition contains no syntax to indicate at which stage the various arithmetical operators should be executed. Nonetheless this is a multi-stage program, whereby the coefficients `a * a` and `b * b` are computed at the current stage and the remaining arithmetic is staged. This is achieved solely by the means of the type declarations: examining the type declarations for the function arguments we find that `a` and `b` are declared to have integer type `Int`, however the types of `x` and `y` are declared to be of type `Rep[Int]`. These type declarations direct LMS to stage all operations within the function body for which the result depends on the integer value of `x` and `y`.

The generic type `Rep[T]` is named to suggest its interpretation as a *representation* for objects of some type `T`. By representation it is meant that an object of type `Rep[T]` at the current stage represents an object of type `T` within the subsequent stage.

Since our declaration of `x` and `y` as having type `Rep[Int]` implies that their integer values are not available at the current stage, and consequently LMS stages any computation depending on those values—the result of the function depends on this staged computation it must be declared as returning type `Rep[Int]` similarly. The available staging operators, policies and optimisations to be applied by LMS are controlled via the type declaration for `this`, which here requires the `Arith` trait (or subclasses thereof) providing support for staged arithmetic operations.

Within its implementation, LMS makes extensive use of language virtualisation. In this example, the `Arith` trait provides the concrete `Rep` type, and overloads its infix `+` and `*` operators. This allows LMS to generate the requisite staged code when the function `polynomial` is invoked, simply as an ordinary Scala function invocation. Through extensive use of operator overloading, and mandating the manual modification of various variable type declarations, the approach coerces native Scala functions into disclosing fragments of their behaviour to the LMS library when invoked.

Preservation of computation order

Naïvely one might expect that LMS separately records the definition of each staged value, such that in our example the return value of the function would simply be defined as the product and sum of its constituents, without any specification of the order in which each of the independent multiplications must be computed. However, LMS automatically preserves the relative order in which operations are computed within staged code, precisely following Scala's own evaluation order. This ordering is then enforced within generated code, typically by generating code in *administrative normal form* (ANF) whereby each operation is ordered formally as a sequence. The ANF generated code for our polynomial invoked with 5 and 6 for a and b would be equivalent to the following fragment:

```
val v0 = 25
val v1 = v0 * a0
val v2 = 36
val v3 = v2 * a1
val v4 = v1 + v3
v4
```

The specific technique of inserting named bindings for each intermediate value in the computation is referred to as *let insertion* [RO10]. Let insertion is a discipline for maintaining the order of side-effects, and reducing re-computation in generated code, by inserting "let" bindings, in order to bring the code into administrative normal form (ANF). For a detailed introduction to this technique see [Rom16]. Here we assume this generated code will be executed within a context binding `a0` and `a1` to the concrete values of x and y respectively, and it is apparent that the multiplication by a^2 happens before the multiplication by b^2. Multiplication of integers in Scala is side-effect free, so preserving the order of computation is immaterial for our example, but given that LMS preserves the relative computation order of operations, it is safe to use side-effecting operations within staged code.

This ordering property of LMS stands in contrast to multi-stage programming languages like MetaML, in which staged code is spliced in a syntactic fashion. Whilst a syntactic approach supports various powerful code generation patterns, it arguably requires more discipline from the programmer. In offering staging operators that abstract away the issues of effect ordering and code duplication, LMS frees the programmer from addressing such concerns manually [Rom12].

Inspection and representation of staged code

In systems like MetaML where staged code is opaque and cannot be inspected, its representation is immaterial to the programmer: since the staged code can only be composed or executed, the formal representation of that code could be encoded in any language supporting the requisite composition and execution operators. In LMS however, the underlying representation of staged code may be exposed selectively, for inspection within certain nominated scopes. The programmer is advised to prevent the parts of the program which are themselves the subject of staging to access their own representation, since that would allow for their behaviour to depend on structural properties of their own representation, and in turn invalidate any staging optimisations that preserve semantics, but do not preserve structural equality of the staged code.

Having made the representation of the staged code accessible, certain possibilities are immediately evident. The ability to inspect staged code offers various opportunities for pure analysis, but also the possibility of offloading execution to heterogeneous or remote hardware by serialising its representation. Although the issues of representation and inspection are potentially separable, it is often advantageous to select a representation which supports tractable inspection for the kinds of analysis or serialisation desired.

Common subexpression elimination

Although the automation of common subexpression elimination (CSE) within LMS might appear as simply a pleasing optimisation, its presence bears directly upon the category of programs which are staged effectively and efficiently. Similarly to how the presence of tail-call optimisation allows for programs to be written in tail-recursive style (which would otherwise be disregarded as impractical) the presence of CSE within LMS avoids the need for certain programs to be written in continuation-passing or monadic style, which would otherwise give rise to staged programs which involve an impractical degree of recomputation.

Applications

LMS has been applied to various problems in which the choice of representation for staged code constituted an important part of the solution. The domain-specific language (DSL) framework Delite [BSL$^+$11], which builds on top of LMS, has been used to implement a DSL for machine learning, OptiML [SLB$^+$11]. The OptiML compiler performs domain-specific analyses and optimisations on its intermediate representation language, before translating to CUDA code suitable for execution on a GPU.

Other notable applications of LMS include: (i) optimisation and translation of SQL queries to C [SKK18]; (ii) enabling higher-level abstraction over contemporary FPGA high-level synthesis tools [GLN$^+$14]; (iii) affecting domain-specific optimisations for mathematical linear transforms, for example the discrete Fourier transform [ORS$^+$13]; (iv) implementing DSLs capable of generating code optimised for execution on heterogeneous computing environments [SGB$^+$13]; (v) accelerating distributed data processing queries [ETD$^+$18]; and (vi) generating C code that is accompanied with annotations supporting certain kinds of verification, including the verification of memory safety properties [AR17].

The most directly relevant application of LMS to this thesis can be found in Lancet, a framework based on LMS, which Rompf et al. introduced for "surgical precision" JIT compilation [RSB$^+$14]. Lancet encapsulates the multi-level language front-end presented by LMS behind a staged metacircular bytecode interpreter, thus providing many of the benefits of LMS to ordinary, single-level programs. Furthermore, they introduce "JIT macros" to allow the compilation of individual methods to be arbitrarily and separately defined. Lancet's approach is so closely related to ours that we delay its comparison until after we have properly defined late-bound code generation, in Section 6.3.

3.3.4 Multi-stage programming with abstract syntax

In this section we will explore two programming languages, Template Haskell [SJ02] and Converge [Tra05], which whilst following in the multi-stage programming tradition, depart in their representation of code values. Both represent code values as abstract syntax trees, data structures which may be programmatically constructed and inspected—we return to this idea again later, within an hererogeneous context in Section 3.4.2. Although continuing to support traditional staging annotations, additional operators for reification are also introduced, which do not require any modification to the interface or syntax of the reified function definition.

Characterising Template Haskell

Template Haskell is a metaprogramming system for the Haskell programming language [SJ02], which provides a superficially similar code generation front-end to MetaML and MetaOCaml. Despite this superficial similarity, Template Haskell differs substantially from these earlier systems.

First and foremost, Template Haskell is a strictly compile-time code generation system, meaning that code can only be generated and spliced into a module whilst it is being compiled. This is in contrast to MetaOCaml, which performs type-checking at compile-time in order to verify the absence of certain type errors within any generated code, yet performs all code generation itself at run-time. Furthermore, quasi-quotation in Template Haskell is simply one method for generating code—intended as a convenience, rather than as a foundational part of the system—whilst within MetaML derived systems quasi-quotation serves as the only constructor for code values. Code values in Template Haskell are represented as ordinary algebraic data types, reflecting the structure of their abstract syntax, and can be constructed programmatically as well as by quotation. There are a number of important consequences to Template Haskell's use of algebraic data types for representing code, which we will now explore.

Firstly, a wider range of code generators are immediately supported, when compared to MetaML-like languages. There are particular code generation functions which are difficult or impossible to write using quasi-quotation alone. For instance, Sheard and Peyton Jones [SJ02] note that generating code which pattern matches against an n-tuple (where n is parametric) is impossible just using Template Haskell's quasi-quotation mechanism:

```
\n -> [| \values -> case values of (v_0, v_1, ..., v_n) -> v0 + v1 |] -- Invalid!
```

Although we might hope to write the above quasi-quote, it is invalid. Neither the ellipsis, nor the fanciful parametric subscript—supposedly bound to the lexically scoped variable `n`—are features of Template Haskell. Special provision in the language would be required to support them. The intended term can, however, be readily generated via programmatic composition of its abstract syntax.

Secondly, code values are no longer opaque, as they are in MetaML derived systems. It is therefore possible to pattern match against Template Haskell's code values, in order to analyse or translate their syntax or behaviour. This enables a wide range of rewrite-based transformations which may be applied to reified or generated code, such as code motion and peephole style optimisations. Although it is occasionally possible to write optimal code generators without recourse to optimisation via post-processing [KST04], it is often exceptionally convenient to be able to apply such optimisations via rewriting of the generated code. In Template Haskell there are, however, a couple of disadvantages to composing code values via data type constructors: (i) the verbosity of the code generator tends to increase substantially, and (ii) the lexical scoping of identifiers is no longer enforced, as it is within quasi-quotes.

This ability to pattern match against code values attains even more power when coupled with Template Haskell's additional reification facilities, an ability which neither MetaML nor MetaOCaml possess, and to which we turn our attention now.

Reification without annotations or interference

Template Haskell provides a selection of additional reification operators: `reifyDecl` for reifying the syntax of a value declaration, `reifyType` for reifying the compile-time type of a value, and also `reifyFixity`, `reifyLocn`, and `reifyOpt` for reifying the fixity of an operator, the current source code location, and command-line options respectively. The first of these `reifyDecl` is of particular relevance to this thesis. The ability to reify the declaration of a function, without any need to annotate or otherwise modify the declaration itself, allows the metaprogrammer to generate code deriving from properties of ordinary function definitions, by pattern matching against the reified declaration. In short, Template Haskell makes it possible for ordinary function declarations to be arbitrarily and programmatically analysed at compile-time, in order to generate translated or optimised definitions, which are then embedded or spliced back into the compiled module. Whilst this combination of capabilities offers much potential, it is neither unique to Template Haskell, nor wholly sufficient, when generalising these capabilities to a run-time context.

For example, Python similarly offers the ability to reify the definition of an ordinary function, although at run-time: a capability we make extensive use of in Chapter 5. When reifying a run-time function—which is in general a closure of the function definition's over the scope in which it was defined—it is also necessary to reify its closure scope, in order to properly analyse the function's behaviour. This ability to reify the closure scope of a function instance is also present in Python—which we build upon in Section 5.4.1—although it is not explored within the literature on Template Haskell, due to its exclusive focus on compile-time metaprogramming. Furthermore, these reification capabilities simply serve as a basis for the analysis, optimisation and translation of functions. The task of performing the translation itself is left unaddressed, as an orthogonal concern. This important task is, however, a primary concern within our late-bound code generation approach.

Notably, later versions of Template Haskell appear to have retired support for reifying the syntax of value declarations. The literature around Template Haskell tends more towards a generative, rather than introspective, approach to metaprogramming. Presumably due to lack of interest and adoption, this specific capability was finally deprecated. As such, there is a lack of research exploring the potential inherent within Template Haskell's original `reifyDecl` facility.

Influence of type checker soundness on design

One major factor which influenced Template Haskell's design was a requirement to support a sound compile-time type checking system. Whilst it is trivial to implement a sound type checker for Template Haskell—by simply applying Haskell's existing sound type checker to the module after code generation has completed—this approach does not lead to type errors being discovered and reported at the earliest opportunity, albeit at compile-time. Furthermore, it is not immediately obvious which code generator is in error when presented with some erroneously generated fragment. For this reason Sheard and Peyton Jones compromised the power of Template Haskell in order to enable more prompt type checking: checking quasi-quotations themselves, as well as their eventually spliced terms.

Later versions of Template Haskell [Jon10] explored even further restrictions, by introducing type declarations to code values. At this point the power of the system was sufficiently compromised that a decision was made to bisect it into two independent versions: (i) a maximally permissive system, even more powerful than Sheard and Peyton Jones' original, which only type checks a module after code generation has terminated, and (ii) a heavily restricted, promptly type checked version, that includes code value type declarations.

Converge as an imperative generalisation of Template Haskell

Because Template Haskell operates within a purely functional context, this introduces a number of technical difficulties into the code generation process. For example implementing "`gensym`", Template Haskell's facility for generating unique names during a code generation phase, requires something akin to stateful effects in order to maintain a history of previously generated names. In characteristic style, Sheard and Peyton Jones solve this by using a monadic device: the "quotation monad". From our perspective these technical difficulties are irrelevant, since we are not restricting ourselves to a purely functional setting. Tratt has explored a metaprogramming system similar to Template Haskell, but embedded within the context of his imperative language Converge [Tra05]. He achieves a similarly powerful metaprogramming system as Template Haskell, whilst doing away with the complications of the quotation monad, and hence making the results of Sheard and Peyton Jones relevant to users of other imperative languages such as Python and JavaScript. Like Template Haskell, Converge represents code values with data types which structurally encode their abstract syntax, and thus supports programmatic construction and inspection of staged fragments. Furthermore, Converge is an inherently more general metaprogramming system than Template Haskell, since it supports code generation not only at compile-time, but also at run-time, thus allowing for code to be specialised to run-time information.

More recently Berger et al. have provided a formalism for relating various homogeneous generative metaprogramming systems [BTU17], encompassing not only Template Haskell and Converge, but also MetaML, MetaOCaml and others. This provides a rigorous basis not only for formally comparing aspects of various compile-time or run-time systems within their own category, but also for comparing aspects of compile-time systems to run-time systems, whilst remaining within the same formal framework.

When comparing Template Haskell and Converge directly to the approach described in this thesis, we note that within these former systems, the kinds of object which are possible to "lift" into generated code are relatively restricted. Only certain specific kinds of value may be lifted within Template Haskell and Converge: in particular, neither systems support lifting mutable objects within generated code. However, such objects are possible to lift using late-bound code generation, an ability which we explore extensively in Chapter 4.

3.4 Type IIb: Self-issued computation from a lexically embedded heterogeneous program

3.4.1 Explicit construction of concrete syntax

The most obvious pattern for self-issuing computation, when dealing with more than one language, is simply to embed fragments of concrete syntax for one language in the other. Those parts of the program which are required to execute within another process—be it an abstract process such as a server thread, or a concrete process such as a physical machine—have their behaviour coded directly in the concrete syntax accepted by that process, and embedded as a string value within the issuing process.

Perhaps the most common, well studied example of this pattern occurs in programs which access persistent data managed by a relational database management system (RDBMS). In this section we describe the motivation, character and challenges presented when applying the pattern in this context.

Programs interacting with an RDBMS typically communicate using a query-based interface. The objective of performing the computation of the query within the management system, rather than within the issuing process, is often a pragmatically motivated: since the RDBMS is typically co-located with the data store, yielding lower round-trip latency, and furthermore obviates the need to communicate any more information over a network than strictly necessary.

Where the query to perform is a constant of the program, many straightforward and safe solutions for providing the query to the database system are available, including defining the query persistently within the RDBMS itself, in which case it is known as a *stored procedure*. When using stored procedures the query can be simply invoked in a manner similar to remote procedure invocation, with only serial forms of the queries parameters and results flowing over the channel connecting the program to the RDBMS. In such situations the interaction of the program with the database systems is no different in nature from that with any other networked service, exposing a *data-oriented* application programming interface (API).

Conversely, in situations where the structure of the query is not a constant of the program, perhaps depending on user input or other runtime information, the form of the query must be generated and communicated with the database dynamically. Queries are typically represented as strings of concrete syntax in some query language, for instance in the ubiquitous "structured query language" (SQL) and exchanged with the RDBMS in this textual form. Although query languages are not necessarily Turing complete, they generally provide operations acting on the database which can violate security properties of the software system they support, for example deletion of data, or reading of arbitrary data, which a particular user is not authorised to access. As such the dynamic generation of queries poses the risk that the generated query might violate important security or safety properties of the system that, which are not immediately apparent from static analysis of the program definition, although fortunately in some cases the requisite static analysis is tractable [WGSD07].

How these risks arise in practice is illustrated by the following Python fragment. The query is built using a string of concrete SQL syntax, dependent on a dynamic dictionary of key-value pairs, which filter the results obtained.

```
predicate = get_dynamic_predicate() # e.g. { 'City': 'London', 'Country': 'UK' }
query = 'SELECT Address FROM Customers WHERE ' + ' AND '.join(
    [key + ' = "' + value + '"' for key, value in predicate.items()]
) + ';'
cursor.execute(query)
for result in cursor.fetchall():
    process_address(result)
```

Depending on the provenance of the dictionary returned by `get_dynamic_predicate`, the program is vulnerable to a SQL *injection attack*. For example if an attacker were able to control the values contained in this dictionary, which might be trivial if they were directly copied from the fields with an HTTP request, then they could easily pass malicious strings such as:

```
'UK"; DROP TABLE Customers; --'
```

Give a sufficiently permissive database configurations this could cause the database to improperly delete important data. Other malicious strings might instead attempt to return more data to the attacker than they are properly authorised to access.

Typically an attack as simple as our illustration is well mitigated in practice, but more subtle variants exist. The practically observed risk of SQL injection attacks, coupled with the technical challenge of generating well-formed query strings, has given rise to a diverse array of mitigations and solutions [HVO$^+$06], of which those language-based solutions range from simple validation and escaping of concatenated strings, to metaprogramming-based code generation solutions comparable to the multi-stage systems described in section 3.3.2 and 3.3.3, for example Cook and Rai's “Safe Query Objects” approach [CR05].

In those cases where the dynamic query is composed in the syntax of another language, as in our example above, we consider this to be an instance of pragmatic control via self-issued computation, from a lexically embedded heterogeneous program, where in many cases the motivation to perform computation within the database can be attributed to pragmatic concerns.

Whilst heterogeneously encoding part of the program's behaviour as SQL strings has historically been a popular approach—since most RDBMS products natively support execution of SQL—somewhat more recently alternative approaches which make use of embedded heterogeneous query languages have been popularised, such as HQL, JDOQL and JPQL [Roo02, GM10, Vas08]. Libraries which automatically translate queries written in these alternative languages to SQL are typically available, for use with a conventional RDBMS. We note that this diversity of embedded query languages does not give rise to a wider range of pragmatic control, beyond that of embedded SQL, but are rather motivated by support for various convenient semantic abstractions: the pragmatic principle is invariably to execute the computation inherent to the query string within the RDBMS, and return any resulting data to the issuing program.

3.4.2 Explicit construction of abstract syntax

The practice of programmatically constructing abstract syntax (outside of compiler implementation) has a long history, extending back to the early days of Lisp. Common practice within Lisp was often constrained to a strictly homogeneous context, within the implementation of macros. In large part this practice was later superseded by quasiquotation, which offers a simpler yet less flexible metaprogramming abstraction [Baw99]. The ad-hoc construction of heterogeneous abstract syntax has in all likelihood a similarly long history, although more recently it has been characterised and formalised within the study of domain specific languages (DSLs) as the category of *deeply embedded* DSLs [GW14].

An interesting and popular contemporary application of this technique can be found within the TensorFlow machine learning library [ABC+16]. Users of the first version of the TensorFlow library must explicitly construct a computation graph, which may be viewed as a form of higher-order abstract syntax (HOAS) [PE88], which represents their desired numerical computation.

For example, consider the following Python program:

```
import tensorflow
a = tensorflow.constant(2.0)
b = tensorflow.constant(21.0)
c = a * b
print(c) # Prints 'Tensor("mul_1:0", shape=(), dtype=float32)'

config = tensorflow.ConfigProto(device_count={ 'CPU': 1, 'GPU': 0 })
with tensorflow.Session(config=config) as session:
    result = session.run(c)
    print(result) # Prints '42.0'
```

Although intended primarily for training and executing machine learning models, here we are using TensorFlow simply as a floating-point calculator.

At first glance the statement `c = a * b` might appear to be numerically multiplying the variables `a` and `b`, but it is an overloaded version of the standard Python multiplication operator: overloaded to act upon and generate computation graphs. The values of `a` and `b` are explicitly constructed as graph nodes which *represent* constant values in the deeply embedded DSL. The overloaded multiplication operator then combines two subgraphs, joining them with a multiplication node to form a larger computation graph. Calling `print` on the value of `c` reveals that it is not Python number, but instead an abstract term of the TensorFlow embedded DSL.

In order to execute the computation graph on the local CPU (of the Python process) a `Session` is configured and instantiated, within which the graph up to and including the multiplication node is run. The result is automatically converted to a NumPy floating-point number and returned—where NumPy is a common Python library for numerical computation—which when printed renders `42.0` as expected.

If we wished to execute the graph on a local GPU, a configuration of `{ 'CPU': 0, 'GPU': 1 }` could be supplied instead. Finer-grained control of the execution target is also supported, including to specific named CPU or GPU units, and remote and distributed systems. This is a form of abstract pragmatic control, whereby the computation is explicitly issued, yet the target processor is defined in part declaratively: `'GPU': 1` specifies explicitly that only one GPU must be employed, but which local GPU if several are available is not specified.

TensorFlow computation graphs are not only subject to optimisation and translation, typically to vectorised CPU or GPU code for execution, but also serve as a basis for other kinds of analysis. An important algorithm in machine learning known as *backpropagation* requires the computation of mathematical derivatives for the embedded program. The computation graph is analysed to provide these derivatives using a technique known as *automatic differentiation*, a numerical method—not to be confused with numerical differentiation—which relies on a non-standard interpretation (with respect to execution) of the computation graph [BPRS18].

Another notable property of TensorFlow computation graphs is their support for variable nodes. Nodes defined using the constructor `tensorflow.Variable` are mutable nodes within the HOAS graph. Graph computations can mutate the value of these variable nodes, with those mutations persisting between individual executions of the graph. This can be understood as a kind of *self-modifying code* within the embedded DSL, specifically as a mechanism for implementing static variables.

The laborious nature and conceptual difficulty (for the programmer) of explicit graph construction have been perceived as usability issues within the TensorFlow community. These issues in part motivated the development of subsequent versions of the TensorFlow library, the second version of which introduced AutoGraph and its accompanying `tensorflow.function` decorator [MDW$^+$19, AMP$^+$19]. Simultaneously, the default semantics of the graph operators were changed to immediately perform their corresponding operation, rather than construct a graph encoding their computation, hence departing from the approach of explicitly constructing abstract syntax, and moving towards an imperative style.

This imperative approach is known as 'eager mode' in the TensorFlow community, and corresponds to a *shallow embedding* in DSL terminology [GW14]. AutoGraph uses reification of the function definition's concrete syntax, amongst several other techniques, to recover the requisite computation graph for functions defined in this manner—transitioning to a type IIa system. The originators of AutoGraph contrast their approach to a complementary progression in the programming model of the PyTorch machine learning library, and its corresponding TorchScript [MDW+19]. Given the extensive user-base for TensorFlow, its evolution provides an interesting case-study for the mainstream acceptance, refinement and subsequent withdrawal of the deeply embedded DSL approach.

3.5 Conclusion

In this chapter we surveyed the background context for this thesis—namely existing intra-program techniques for pragmatically controlling the execution of sub-program fragments. We informally categorised these various techniques into types I, IIa and IIb.

Type I techniques, which are perhaps the most popular, are based on directives. Given the unbounded potential for pragmatic control which is afforded by a directive, we illustrated this category by means of a number of examples: specifically directives for controlling parallelism, applying and defining optimisations, and for language subsetting. Although many other kinds of directive have been defined, we believe that these four examples characterise the domain within which directive-based control has been explored and applied.

Type IIa and IIb techniques are based on self-issued computation, where the former relies on reifying a fragment of the homogeneously defined program, and the latter on explicitly constructing the heterogeneous syntax for the fragment. The division between type IIa and type IIb techniques is imprecise, even more so than the division between type I and II. To a large extent it is subjectively dependent on one's definition of the language semantics in question. For instance, the categorisation of LMS depends on whether LMS is considered a language itself (extending the Scala programming language) or simply a conventional set of objects defined within a Scala process. Nonetheless, we believe that the division between type IIa and IIb captures an important distinction for the programmer: if the illusion that the entire meaning of their program is defined in one language is broken, then the technique is better characterised as type IIb. In many practical circumstances, programmers may have an aversion to multi-language based solutions, since they typically pose a challenge to tooling and integration—for example, consider the problem of how exception stack traces might propagate across language boundaries.

We again illustrated our informal type IIa category by means of examples, including three different approaches based upon quasiquotation, annotation-directed multi-stage programming, and type-directed multi-stage programming. Finally we gave examples of type IIb techniques, based on both the construction of concrete syntax and abstract syntax, the two principal approaches for defining executable forms.

Chapter 4

Low-power user interface rendering

4.1 Introduction

In this chapter we explore in detail one particular application of late-bound code generation: applying it to the problem of offloading the user-interface rasterisation of a smartphone application to a field-programmable gate array (FPGA) coprocessor. We aim to organise our rasterisation circuit architecture in a form well suited to place-and-route style optimisation, in an attempt to reduce its energy consumption. Our intent is to explore how well late-bound code generation—as a technique—fares when applied to such a problem, having externally defined constraints and objectives.

We describe a method for transforming the user interface rasterisation procedure of a smartphone application, defined in terms of both standard and programmatically defined components, to an application-specific statically scheduled pipeline, suitable for execution within a FPGA. The state of the application is synchronised with that of the pipeline via a serial channel. We apply our method to a test application, and estimate the power consumed by the generated pipeline. The potential for our approach to reduce the power consumed within a mobile device is evaluated, along with its inherent limitations.

Since the semiconductor industry no longer enjoys the benefits of Dennard scaling [Boh07], we must look toward architectural solutions if we wish to reduce the power consumed by electronic devices such as smartphones. Reducing power consumption might be motivated by (i) an application feature which requires significantly more computation to be performed, (ii) to reduce the battery charge cycle frequency, (iii) to reduce the weight or cost of the power supply, or (iv) any combination of these factors. In order to approach the problem we must first understand how energy is typically dissipated within a device.

Within a smartphone there are various key components, logical and physical, which together dissipate the majority of its energy. Typically these key components include the radio module, fixed-function display hardware, the user-interface rendering process and finally miscellaneous application-processing operations [CH10]. Depending on the usage scenario any of these four categories can dominate. For example, in an area of poor reception the radio module might contribute most to the power consumption of the device. When operating the device outdoors, the fixed-function display hardware—which includes the hardware for illuminating the screen—might dominate instead. Alternatively, using a mobile device indoors with good radio reception to a nearby wireless router, can result in the user interface rendering becoming the single largest contributor to the power consumed. It is this final scenario which is of primary interest in this chapter, and in particular how the requisite rendering computations might be optimised or offloaded to minimise the total power consumed.

4.2 Contributions

- We describe a method for obtaining an application-specific object processor pipeline for rasterising the user interface of a smartphone application, derived from a source program written in terms of contemporary abstractions, and featuring both standard and custom user-interface components.
- We describe how any parameters within the pipeline that depend upon application state may be discovered and synchronised with the application runtime.
- We provide an account for the power consumed during execution of this pipeline on a

commercially available field-programmable gate array (FPGA).

4.3 Background

Before we can make an hypothesis for reducing the energy consumed during rendering, it is necessary to understand first the logical and physical rendering architecture of a contemporary smartphone, and then which aspects of that architecture dissipate energy unnecessarily.

4.3.1 Logical rendering architecture

Consider the rendering architecture for the Android operating system [LtAOSP22]: we take this as our reference architecture throughout this chapter. On Android, rendering begins with *view* objects—the user interface of an application is composed entirely of views. Views are objects that represent components of the user interface, and are organised hierarchically as a tree. The view abstraction encapsulates several user interface concerns, just one of which is rendering—for example, views also manage the interaction capabilities of the component, mutable state particular to the component type, and layout strategies for aligning the component within the available screen space.

Any view may be rendered by calling its `draw` method. Calling `draw` on a view which has children will normally dispatch further calls, recursively drawing each child. The method takes just one argument, the canvas onto which the view must be drawn. The canvas represents a two dimensional surface, the appearance of which is manipulated via its various methods, such as `drawLine` or `drawOval`. There is considerable freedom in the implementation of the canvas. For instance a particular implementation might directly mutate the state of an underlying bitmap image. Alternatively it might represent the state of the canvas symbolically, for example mutating the elements of a vector graphic.

Early versions of Android implemented their `Canvas` class by immediately dispatching commands, such as `drawLine`, to an underlying rasterisation library called Skia [LI22]. In these early versions, only one CPU-based bitmap rasteriser for Skia was available. As the platform evolved, motivated primarily by the desire for greater performance, the Android developers introduced another canvas implementation: the `RecordingCanvas`. Unlike the original Skia-based canvas, `RecordingCanvas` does not perform rasterisation immediately when its draw methods are invoked, but instead appends each graphical operation symbolically, along with its parameters, to a display list. This idea of a display list did not originate in Android however, and may be traced back at least to the 1979 ANTIC coprocessor for the first Atari console [AI82].

Populating this (symbolic) display list is computationally cheap compared to the rasterisation of each command. Only after the display list has been fully populated, the canvas may be rasterised in batch. The purpose of introducing the `RecordingCanvas` was in part to support offloading the rasterisation process to a GPU accelerator: the display list populated by a `RecordingCanvas` serves as an intermediate representation for the state of the canvas, which may then be optimised by reordering and batching operations, before translation to an equivalent series of OpenGL commands to be dispatched to the GPU. It is worth emphasising that each time the screen is updated, up to 60 times per second, the display list is at least in part re-generated, re-optimised, re-translated, and re-issued to the GPU. Portions of the display list are opportunistically cached, along with bitmap regions of the canvas, but the major class of optimisations enabled by the display list abstraction are intra-frame optimisations.

This application of intra-frame optimisation, in combination with the increased parallelism of the GPU architecture, allows for a significant reduction in the time taken for Android to render each user interface frame.

From an energy perspective however, there are outstanding inefficiencies, in both the original CPU-based approach and the latter GPU-based approach. CPUs and GPUs are fetch/execute machines which dynamically multiplex their various resources, communicating intermediate results to and from register files and caches, which involves energy wastage. Furthermore, the employment of display lists introduces additional computation due to re-optimising and re-programming the GPU whenever the frame is updated—the energy cost of which might be reduced if it were amortised over several frames.

4.3.2 Accounting for energy expenditure

The internal physical structure of a computer must be understood in order to account for the energy consumed during its operation. Contemporary computers, including most microprocessors, graphics accelerators and certain application-specific integrated circuits, are typically implemented as fetch/execute machines, fabricated using a complementary metal-oxide-semiconductor (CMOS) process. In a fetch/execute architecture, separate physical regions are set aside within the silicon wafer for the storage of instructions, for the storage of data such as register files, and further regions for performing arithmetic and logic calculations (ALU). The processor operates in a cyclic manner, processing instructions approximately one after another, inspecting each instruction to determine what data must be read, what calculation to issue to the ALU, and finally where the result must be written. Certainly many modern computers are somewhat parallel in their operation, in that they are able to process several instructions at the same point in time, for example utilising pipelines or duplication of the fetch/execute engine, but from an energy perspective there are inefficiencies inherent to their physical organisation. To understand this we must inspect the fetch/execute cycle in more detail.

Each fetch/execute instruction by design requires information to be physically transferred between the computer's instruction cache, register file, and ALU, and then back again, every cycle. On certain occasions this excursion might be shortcut, due to forwarding of values between stages of the processor's instruction pipeline, but this is an opportunistic optimisation. In either case the ALU acts as a shared resource multiplexed between different instructions. This strict partitioning of silicon between data storage and calculation was quite necessary historically, since on older silicon processes the physical area required to implement an ALU was far from negligible. However the area saved by multiplexing a small number of ALUs between a larger number of instructions is traded for an increase in data movement. If instead the ALU was massively replicated, with the output of one fed directly into the input of another, perhaps each ALU dedicated to one individual program instruction, the amount of data communication would be significantly reduced: for it is communication which costs considerable energy in a CMOS computer.

Communication costs energy because CMOS processors dissipate energy whenever current flows though their transistors. That current can be broken down into leakage current and switching current. The leakage current per transistor is approximately constant for a given CMOS process and supply voltage, and can only be lessened by reducing the area of the silicon wafer receiving voltage bias, or by improving the silicon process itself (also a prerequisite for reducing the voltage). Switching current on the other hand occurs whenever the input to a transistor is inverted. The amount of charge which flows, and so its cost in terms of energy, is proportional to the capacitance at the transistor's output. The longer or larger the output traces are, the larger the capacitance—in other words the further the output of the transistor must be communicated the larger the energy cost. Thus, there are fundamentally two ways to reduce the power consumed due to switching, reduce the total number or frequency of signal inversions (i.e. perform fewer or simpler macro- or micro-operations) or, reduce the length over which transistors must communicate.

It so happens that in contemporary processors the amount of energy spent communicating data between caches and register files vastly outweighs that expended switching, at least for most ALU operations. For instance on a 45 nm CMOS computer only approximately 0.14% of the energy consumed by a 32-bit integer addition is spent inside the ALU, whereas some 40% is spent on cache and register file access [Hor14]. A significantly more complex operation such as a 32-bit floating-point multiplication still expends only about 5% of the total instruction's energy within the ALU. This assumes that the data is in the register file or data cache—if there is a cache miss the requisite access to dynamic read-only memory (DRAM) likely further reduces these percentages, by at least a factor of 10.

But still, this leaves less than 50% of the energy accounted for. We must also address where the majority of the energy is spent. According to Horowitz [Hor14] the majority of the energy spent executing an instruction is dissipated in control logic. This might be anything from the simple logic which controls an in-order scalar processor pipeline, to the elaborate speculation logic found in modern out-of-order super-scalar processors—they do not elaborate, but we suspect the latter. Nevertheless a good deal of energy is potentially spent on the task of dynamically multiplexing CPU resources, such as the ALU, in addition to that spent on internal communication.

In summary there are four primary categories of inefficiency when accounting for energy expended in a CMOS based computer, be it a CPU, a GPU or any competing ASIC architecture:

- Static leakage current (smaller transistors / fewer power domains implies more waste)
- Internal communication of information (greater distance implies more energy dissipated)
- Dynamic organisation of resources (for example multiplexing shared ALU / speculation)
- Unnecessary computation (algorithmic inefficiency or failure to remove abstraction)

4.4 Hypothesis

Given our abstract characterisation of Android's rendering architecture, and this fundamental appreciation for energy dissipation with a CMOS device, where might energy be wastefully dissipated in contemporary smartphones, specifically when rendering their user interface?

By observing how the user interface changes from frame to frame, it is manifestly apparent that in typical applications the variability is relatively constrained. Often frames share exactly the same selection of user interface components, thus the same views, and consequently the same underlying graphical primitives: each frame is largely a *rearrangement* of the previous frame, sharing the same primitives, but with different coordinates and parameters.

This means that the control-flow of the *rendering* process is, to a large extent, static. In certain cases the extent to which the control-flow is static is a literal consequence of certain terms within the application source code, whereas in other cases it is implicit: for instance a particular view might always draw three lines, explicitly witnessed by three statements calling `drawLine` on the given canvas, within its `draw` method; alternatively a particular container view might *support* dynamic removal and addition of children over time, but two specific views are only ever added during construction. In this latter case the static extent of the control-flow is implicit in the application source program.

If this observation holds, at least for the major proportion of frames based on typical usage patterns, then it would follow that the energy spent performing dynamic control-flow within the silicon device is largely wasted. It is not *necessary* to dynamically multiplex the input and output of the ALU within the CPU or GPU every cycle, nor to store intermediate values within register files, since the flow of values from one operation to the next is predetermined. The energy dissipated in both the control-flow circuitry and the additional current sourced / sunk driving high-capacitance lines to common storage could in principle be eliminated.

If a (pure) data-flow graph for the rendering process, specialised to a given application, could be obtained, then a radically more efficient circuit with fixed control-flow could be constructed, to take over from the GPU when applicable. This circuit must be specialised to the application in order to be efficient, since the control flow for the rendering process is typically static only within the context of a given application, and not between applications: when switching between applications the set and draw-order of primitives making up the user interface changes significantly. It is of course true that most applications display several distinct modes of user interface presentation within themselves, which for simplicity we are ignoring, but the approach of specialising a circuit to an application applies equally to sub-modes of an application.

4.4.1 Object-oriented rasterisation

Within the computer graphics literature, rasterising an image in the manner just described is referred to as *object-oriented rasterisation*, and more specifically as *processor-per-polygon rasterisation* [GGSS89]. The specific term *processor-per-primitive pipeline* [FvDFH96] also well describes the architecture we employ, and we follow Schneider [Sch88] in using the term *object-processor pipeline* (OPP) to identify the circuit which carries out the rendering process. Early examples of processor-per-primitive pipelines were motivated less by their potential power-saving advantages, but instead their increased parallelism and reduced silicon area requirements compared to framebuffer-based techniques: they are capable of delivering a pixel per clock-transition, and on demand, without the need for voluminous buffer circuitry.

The basic idea is that within an OPP each individual primitive to be rasterised is allocated (although not necessarily permanently) to a particular *region* of the circuitry. The sub-circuits allocated to each primitive are physically connected together in a draw-ordered linear sequence, and a stream of raster-ordered pixel values are pipelined through the circuit to produce the pixel stream of the rasterised image. Pixel values typically consist not solely of colour information, but also geometric information such as raster coordinates and other miscellaneous symbolic metadata.

Whilst implementing a *generic* OPP directly in silicon confers many of the benefits of a processor-per-primitive pipelined architecture, a generic implementation negates many of its potential power-saving advantages by failing to elide much of the dynamic control-flow circuitry. This is particularly acute if more than one type of primitive is supported by the pipeline, since the pixel signals must be driven and routed around unused processor units. In contrast to previous implementations of processor-per-primitive pipelines, we instead synthesise an application-specific OPP from within the application runtime, and implement the pipeline on an FPGA. The advantage of this approach is that the synthesised OPP contains minimal control-flow circuitry and explicit data-flow dependence.

The explicit data-flow graph presented by the synthesised OPP may be consequently fed into a place-and-route algorithm, in order to minimise the energy dissipated in the circuit due to communication: the place-and-route algorithm is capable of optimising the physical circuit layout in order to reduce the wire length between connected circuit components. This *not only* minimises the capacitance and thus power dissipated due to the interconnect, but also maximises the allowable clock frequency, which as we shall see later, also leads to a reduction in static power dissipation.

Although this approach results in an OPP with minimal control-flow overhead, and is amenable to place-and-route optimisation for low power—the cost of which might be amortised over many thousands of frames—our decision to implement it within an FPGA introduces a considerable quantity of multiplexing transistors into the final circuit. Because of their reconfigurability, implementing a circuit within an FPGA is mediated by transistors which would otherwise be absent within a fixed-function ASIC. This necessarily results in not just higher capacitance on the output terminals of the switching transistors, compared to an equivalent circuit synthesised to an ASIC, but also considerably higher static leakage current due to the biasing of the transistors inherent to the FPGA fabric.

The immense opportunities within place-and-route optimisation are tempered by the inherently high cost of performing such optimisations, and so must be amortised over a correspondingly large number of executions. This leads to the major challenge of obtaining a *useful* OPP: although the OPP must be specialised to the application, it must be generic enough to describe the rendering process for a long enough *sequence* of frames to marginalise the place-and-route optimisation cost. Since the variability within the frame sequence is a function of application state, the OPP circuit must present a port through which certain internal registers may be synchronised with the state of the application.

Since such an OPP must render its view in not simply one specific state, but rather within a particular domain of the states, this yields what is a *metaprogramming* challenge—the generic behaviour of the view's `draw` method must be captured within a context of certain indeterminate variables or bindings.

4.5 Object-processor pipeline architecture

The application-specific OPP generated by the method we present is architecturally an acyclic graph. In general, nodes within the OPP do not share a common interface, but all nodes directly responsible for rasterisation of the pixel stream must at least extend the *raster node* interface. The output node of the OPP is always a raster node.

For illustration, a Verilog module for rasterising an axis-aligned filled rectangle, conforming to the raster node interface, is given as follows:

```
module FillAlignedRectangle(
  input Clock,
  input Clip,
  input [55:0] InputPixel,
  input signed [15:0] VertexX,
  input signed [15:0] VertexY,
  input signed [15:0] Width,
  input signed [15:0] Height,
  input signed [15:0] FillR,
  input signed [15:0] FillG,
  input signed [15:0] FillB,
  output reg [55:0] OutputPixel
);

  wire signed [15:0] ApertureX = InputPixel[55:40];
  wire signed [15:0] ApertureY = InputPixel[39:24];
  wire [7:0] InputR = InputPixel[23:16];
  wire [7:0] InputG = InputPixel[15:8];
  wire [7:0] InputB = InputPixel[7:0];

  wire InsideRectangle =
    ApertureX >= VertexX &&
    ApertureY >= VertexY &&
    ApertureX < (VertexX + Width) &&
    ApertureY < (VertexY + Height);

  wire [7:0] WorkingR = (InsideRectangle && !Clip) ? FillR[9:2] : InputR;
  wire [7:0] WorkingG = (InsideRectangle && !Clip) ? FillG[9:2] : InputG;
  wire [7:0] WorkingB = (InsideRectangle && !Clip) ? FillB[9:2] : InputB;


  wire [55:0] WorkingOutputPixel =
    {ApertureX, ApertureY, WorkingR, WorkingG, WorkingB};


  always @(posedge Clock) begin
    OutputPixel <= WorkingOutputPixel;
  end


endmodule
```

All raster nodes take as input: (i) an edge-triggered clock shared between all nodes; (ii) a rasterisation clipping signal; and (iii) an input pixel stream. They produce as output a (registered) pixel stream. The input pixel stream delivers pixels of the image onto which the node rasterises its primitive. The output pixel stream delivers the image, which has been composited with the primitive of the node, to further raster nodes—unless the node in question is the ultimate output node of the OPP. Raster nodes may also take additional node-specific inputs and outputs, such as in this example parameters which define the geometry and colour applied to the rectangle. All nodes within the OPP must have a static latency between the input pixel stream and output pixel stream: this particular node has been designed with a fixed latency of 1 cycle. This fixed latency constraint is the primary reason that our OPP circuit can elide control-flow logic, since it allows the computation to be fully statically scheduled.

The pixel stream is a 56-bit wide data-path, which delivers a single pixel of the image every clock cycle. The lower 24 of those 56 bits represent the colour of the pixel using 3 bytes, one for the red, green and blue colour channels. For simplicity we elide an alpha channel from the pixel stream, but that does not prevent each node from internally alpha blending the results of their rasterisation with the input pixel stream. The upper 32 bits represent the coordinates of the pixel as a pair of signed fixed-point numbers, enabling rasterisation within 16 sub-pixel regions.

Implementing an axis-aligned rectangle rasteriser according to this interface is quite simple, as shown above. Each clock edge delivers a new pixel to the input of the node, which is destructured into its coordinates and colour channels. These values are then fed into adders and comparators to determine whether the pixel lies inside or outside the rectangle. If this condition is true, and that part of the rectangle is not clipped (invisible) then the fill colour of the rectangle replaces the colour of the input pixel, otherwise the pixel is unchanged. These calculations are allowed one clock period, minus the setup time of the output pixel register, to settle. On the rising edge of the next clock transition, the output pixel value is captured by the output register, and held for subsequent nodes to compute upon. It is more efficient to register the outputs of a node rather than its inputs such that branches in the OPP do not introduce redundant registers.

Implementing nodes for other graphical primitives might involve more complex logic, but the task is well defined, and without the fixed latency constraint is clearly possible, if not necessarily maximally efficient with respect to other rasterisation architectures. Primitives which require unbounded iteration pose a challenge in assigning a static worst-case latency, but such primitives are not typical within user interface rasterisation. In such cases where an OPP is not suitable for rasterising the entire image, it can be composited with a complementary image generated by another processor, such as a CPU or GPU.

As well as performing place-and-route optimisations, the synthesis tool which implements this register-transfer level (RTL) description of the OPP into the FPGA fabric may also perform aggressive constant propagation and folding optimisations on the circuit. For instance in this example of an axis-aligned rectangle, if the vertex coordinates are given by a literal value within the OPP then the comparators may be specialised to that value, leading to potentially far fewer transistors in the implemented circuit. This is another form of optimisation which is enabled by the OPP representation that can reduce the power consumed by the rendering process.

In the simplest of OPPs the raster nodes form a linear graph, with the output pixel stream of one node fed directly into the input pixel stream of the other. Parameters to the raster nodes are supplied by various *value nodes* within the OPP, which feed directly into each raster node. Any parameters which are defined constant within the RTL are typically folded into the raster node during circuit synthesis. Parameters which are not constant, but depend on the state of the application runtime, are supplied by registers which are synchronised with the application state during the inter-frame period. The specifics of this synchronisation mechanism are discussed later in Section 4.8.2.

More complex OPPs involve branching graphs, where the pixel stream is fed into several downstream nodes, to be later combined. This is especially common when primitives are clipped within a region defined by a mask. The nodes responsible for clipping computations culminate in the production of a boolean signal which controls the requisite clip input to all raster nodes. This signal is used to selectively disable a node's effect on a per-pixel basis.

In order to allow values to diverge and combine in a statically scheduled pipeline, care must be taken to ensure that when values from different paths combine at a common node, the values are synchronised with each other. Failure to do this, whereby values pertaining to *different* pixels collide at a single node, usually results in corruption to the raster image. We introduce fixed-latency queues into our OPP to ensure that this cannot happen. The algorithm is straightforward: when building the OPP, which as we shall see is assembled recursively from fragments, the maximum latency across each input fragment is calculated (taken back to the origin of the pixel stream) and then queues are inserted before each input to ensure that the latencies are exactly equal. Queuing circuitry is the price paid for static scheduling, and we hypothesise that the energy expended on the queue transistors is lower than what would be spent on dynamic control-flow circuitry.

4.6 Experiment

In order to test our hypothesis we created a simple smartphone application, from which an OPP was generated and implemented on a commercial FPGA. The user interface of this test application is simpler than typical for a smartphone application, yet contains challenges representative of the state of the art in user interface design: specifically the central feature of our test application is what is known in Android as a *recycler view*. A recycler view is one of the most complex standard components provided on Android. It contains a notionally infinite sequence of child views, laid out either vertically or horizontally, clipped within a rectangular region. Users scroll through the contained views, usually by swiping their finger repeatedly, to bring obscured views into the visible region whilst pushing other views out. In practice the entire sequence of views, both visible and hidden, are *not* materialised within the runtime, but as views disappear, becoming fully clipped, they are recycled as incoming views, an implementation detail from which the recycler view gets its name. This optimisation is effective since the child views of a recycler views are often all instances of the same class, but just differing in the contents of various fields. Furthermore views within the recycler view are typically instantiated dynamically, whereby data required to instantiate each child is loaded just before it becomes visible. This subtle combination of dynamism, heavy reliance on clipping, and careful optimisation makes the recycler view an interesting test case for proving the capabilities of our OPP architecture.

For these reasons our test application contains a prominent recycler view, along with a title view and one other kind of view known as a "floating action button", which partially covers the recycler view. All of the children of the recycler view are instances of the same class: a custom view consisting of two text views. Due to the recycler optimisation, only four instances of this class are ever instantiated, the maximum number which could ever be visible concurrently: these four instances form a pool from which the children are recycled, as the user scrolls through the content. A screenshot of this test application is given in Figure 4.2.

4.6.1 View definitions

This hierarchy of views comprising the top-level of our application is defined, as in Android, within an XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<ConstraintView
```

```
  Left="0" Top="0" Right="640" Bottom="1152" Background="#E0E0E0"
  FixedDimensions="true" FixedBackground="true">
  <TextView
    ID="MainTitle" Text="My I/O"
    Left="32" Top="50" Right="608" Bottom="148" Background="#00E4FF"
    FixedBackground="true">
  </TextView>
  <RecyclerView
    ID="MainRecycler"
    Left="32" Top="198" Right="608" Bottom="1086" Background="#FFFFFF"
    FixedBackground="true"
    UniformChildView="ScheduleItemView" MaximumChildCount="4">
  </RecyclerView>
</ConstraintView>
```

The custom view `ScheduleItemView` is defined separately, again in XML, as:

```
<?xml version="1.0" encoding="utf-8"?>
<ConstraintView
  Background="#F0F0F0"
  FixedBackground="true" FixedChildSignature="true">
  <TextView
    ID="SlotTitle" Text="Loading..."
    Left="32" Top="50" Right="544" Bottom="148" Background="#1CE8B5"
    FixedBackground="true">
  </TextView>
  <TextView
    ID="SlotDescription" Text="Loading..."
    Left="32" Top="148" Right="544" Bottom="246" Background="#E0E0E0"
    FixedBackground="true">
  </TextView>
</ConstraintView>
```

There are several attributes within these XML definitions which make use of features not available in standard Android. We introduced these extensions to enable automatic and efficient generation of the OPP. The novel attributes are as follows:

- `FixedDimensions`
- `FixedBackground`
- `UniformChildView`
- `FixedChildSignature`
- `MaximumChildCount`

The first two attributes are simple constancy annotations: they allow the user interface designer to convey that the dimensions (width and height) and background colour of a particular component are constant respectively. Hence the dimensions and colour of the view may be inlined and constant folded within the OPP during synthesis, and hence do not require synchronisation with the application runtime, conserving both time and energy.

The `UniformChildView` and `FixedChildSignature` annotations on the other hand are a kind of type annotation. The former allows the user interface designer to state that all the children added (perhaps dynamically) to a parent view will have a particular concrete class. The latter annotation states that the list of each child's concrete class is constant, but not necessarily uniform. Finally `MaximumChildCount` places an upper bound on the number of child views which may be added to the given parent view.

Without these additional annotations to the view definitions it would not be possible to automatically generate an efficient OPP. The default behaviour of an Android `RecyclerView` permits any type of view to be added as a child at any time. Without foreknowledge the generated OPP must necessarily accommodate any such possibility. In general this is not possible within a statically scheduled pipeline—and even if a generous and speculative upper-bound were placed on the latency, the complexity of the general purpose circuitry required would likely negate the power savings intended. Hence it is important when generating the OPP to have some bound on the type of views which may be present within the recycler view. The simplest bound, which we use in our example, is the singleton set: we constrain the recycler view in our test application to children with exactly the concrete class of `ScheduleItemView`, a custom application-specific view class. For the same reason it is important to place a bound on the number of children, which in our example is manually set to four. This bound might have instead been inferred automatically, if sufficient dimensions pertaining to the recycler view and its children were defined constant.

Consequently the section of the OPP responsible for rendering the recycler view need only contain four copies of the OPP fragment for rasterising a `ScheduleItemView`, which may be selectively disabled via their clipping signal whenever fewer than four children are concurrently visible.

The `FixedChildSignature` annotation on the recycler view's children might initially appear redundant, but is required for the same reason as the annotations on the recycler view itself. This is because the XML definition of a view in Android is only a template for constructing the view, a process known as *inflation*, not an atemporal definition. In general views may be arbitrarily mutated after inflation, to the extent permitted by their methods. Consequently these annotations are employed not solely as compile time metadata, but are enforced at runtime—any attempt to mutate the state of a view in such a way as to violate the annotation leads to an immediate runtime exception being raised within the offending method. Without the `FixedChildSignature` annotation it would not be possible during the generation of the OPP to rely upon the fact that each recycled child consists of exactly two text views. Furthermore views need not be constructed from XML, even in the first instance, but might originate from procedural code. Nevertheless views which originate procedurally can be constrained by calling similarly named methods which correspond to these annotations, for example a `FixBackground` method might stand in for the `FixedBackground` annotation.

4.6.2 Data dependence

In Android the XML description of an application's user interface does not completely define its appearance and behaviour. This is because the appearance of custom views can be defined within the application *program* by overriding their draw method. An additional complication which must be handled is that views which depend on application state are typically mutated from time to time, as the application evolves. In our example the recycler view depends on application state in two different ways. First the scroll position of the view depends on interrupt driven interactions from the user, and second the text displayed within each `ScheduleItemView` depends on data acquired by the application, perhaps over a network connection. These dependencies are not even specified within the XML definition of the user interface, but are established within the application program.

When generating an OPP specialised to an application, these kinds of state dependence must be discovered and synchronised with the application as the state evolves. Given that a single frame of the user interface pertains to a single instant in time, there is no need to synchronise state with the OPP whilst the pipeline is executing. Instead the state of the OPP may be updated only between frames, when the pipeline is inactive.

The approach taken is to add each view which forms part of the user interface, during OPP generation, to a set known as the *preemption set*. All objects within the preemption set are candidates for state dependence: objects within the preemption set are serialised just before a frame is rendered, and the serialised forms are transmitted to the OPP in the inter-frame idle period. The set is termed a “preemption” set since the state of the object is bound earlier than it would be when rendering the same view natively on the CPU. Therefore there is a danger that this could lead to an incorrect rasterisation result: if the state of a preempted view changes in the intervening time, namely during the rasterisation process, then attributes of the preempted state could differ from the correct value. Although mutation of views during rasterisation is informally discouraged within Android applications, the possibility still remains. Therefore during the code analysis performed during OPP generation we mitigate the hazard by *formally* checking that no state mutation operations occur during rasterisation.

It is of course possible for the OPP to also have a state dependency which is not via a view object of the user interface: in this case the object is included in the preemption set and synchronised in exactly the same manner as a view object.

4.6.3 Circuit synthesis

Having supplied annotations to the user interface definition, which place bounds on its variability, and having a strategy to discover its dependencies of application state, an OPP for the application may be generated from its top-level view object. Exactly how this is done is discussed fully in Sections 4.7 and 4.8, but abstractly the process proceeds as follows.

When the top-level view of the application's user interface is bound to its window, the OPP is immediately generated and installed into the FPGA. This binding usually occurs just once, when the window is created, and so one OPP installation serves for the entire lifetime of the application. In order to obtain a specialised pipeline, the behaviour of this top-level view's `draw` method is translated into an acyclic graphical representation of the OPP. This representation might be thought of as a RTL intermediate representation for the target circuit: it comprises the logic of each circuit component, and their relations, but does not commit to a specific FPGA block-level implementation, or any placement decisions. This graph is trivially cast to a netlist, in either Verilog or VHDL, since both representations are at essentially the same level of abstraction—our graphical representation simply consists of additional metadata over the underlying Verilog form.

This intermediate representation for the OPP, whilst not committing to a particular gate-level implementation, does commit to a particular timing schedule. This is motivated by our explicit aim of removing all dynamic control-flow from the resulting circuit—by fully inlining all method calls during OPP generation, we explicitly obtain a statically scheduled pipeline. This naturally prohibits *unbounded* iteration and recursion within the rendering process, but their presence within the rasterisation process is discouraged in any case, since the process must complete within a fixed interval. It is not uncommon for application developers to write what is effectively bounded iteration using an unbounded form: the annotation mechanism described previously allows for these implicit bounds to be supplied to the generation process.

Wherever values converge in the OPP, queues are inserted as necessary, such that all nodes need take only one control signal: the common pipeline clock. Additionally to the rendering pipeline, the OPP contains an interface for controlling its execution and receiving its output. This interface includes:

- An input signal for the pipeline's clock;
- Input signals for the raster scan coordinates stream;
- Output signals for the raster scan coordinates and pixel stream;
- The pipeline latency (measured in clock periods);
- An interface for updating the pipeline state from the serialised preemption set.

Having cast our OPP representation to Verilog, it is possible to delegate the task of implementing the FPGA circuit to one of several third-party tools. Most FPGA vendors provide custom tool-chains for implementing a circuit within their products, operating from a definition in Verilog.

This process consists of several sub-operations, including but not limited to: synthesising logic gates from the RTL description; mapping and packing gates into the vendor's fixed-size lookup tables (LUTs) and flip-flops; placing individual LUT instances within the FPGA fabric; routing connections between blocks; and performing timing analyses to determine bounds on both the maximum clock frequency and power consumption. In particular the time taken to place and route the circuit for the FPGA is often in excess of minutes, even approaching hours. Fortunately the OPP generated by our method is typically stable between instances of the same application, and thus the substantial energy and time cost of running this tool-chain could in principle be shared between application instances, via either a local or remote cache.

Due to the large number of transistors used within any FPGA to implement the reprogrammable circuitry, they usually exhibit considerably higher static consumption than an equivalent circuit implemented in an ASIC. Whereas a simple gate within an ASIC design might require less than a dozen transistors, and with direct wiring employed for its fan-out connections, the same gate implemented within an FPGA might be implemented by writing state into a static random-access memory (SRAM) LUT with indirect connectivity via the FPGA matrix.

The static power consumed by the FPGA transistors can be considerably reduced in two ways: LUT technology selection, and effective power gating. Most commercial FPGAs use SRAMs to implement their LUTs, but there are alternative products which use Flash memory as their LUT technology. Flash memory based FPGAs can consume several orders of magnitude less static power [MTIMPP22]. This is due in part to their non-volatility and reduced transistor count. Power gating is another highly effective mitigation for static power. The challenge in applications such as rendering, where calculation phases occur with some tens of milliseconds periodicity, is in powering the accelerator down and back up again within the available idle window, and furthermore without expending so much power during the power transition as to negate any benefit. For this reason we selected an FPGA manufactured by Microsemi, from their “PolarFire” range—a series of Flash based devices which can enter and leave their power gated state within approximately 200 microseconds [MTIMPP19]. This amounts to less than 2% of the inter-frame period for a 60 Hz refresh rate, a small enough fraction for it to be practicable to power the FPGA up and down every frame. Microsemi’s power gating technology, marketed as “Flash*Freeze”, does not fully power down the FPGA, but gates power to the *majority* of the device in order to reach a compromise between static power reduction and the latency of the power gate.

To minimise the average power consumed by the FPGA it is potentially necessary to pipeline individual nodes within the OPP, increasing their latency in cycles whilst reducing the length of the circuit’s critical path, so attaining the highest possible frequency and thus the longest possible operation interval with the power gate open.

Figure 4.1 shows a schematic for the OPP generated by our experimental application, described in section 4.9. Whilst the precise details are not important at this point, it can be understood from a high-level perspective as globally clocked pixel pipeline (with associated metadata) flowing from top to bottom. Pixel coordinates are fed into the top-end of the pipeline, one $\{x, y\}$ pair every clock cycle. Rasterised pixels then emerge from the bottom-end of the pipeline after some latency, one pixel per clock cycle.

4.7 Late-bound pipeline generation

It is tempting to imagine that an OPP for rendering the user interface of an application could be simply generated directly from an XML definition. But as we have seen in Android, the XML definition serves only as a set of construction parameters, lacking any information about how the user interface evolves with application state and the user's interactions. Furthermore since the user interface might contain programmatically-defined custom views, it is necessary to generate the OPP from the object structure of the program itself.

Whilst it might be possible in principle to generate an OPP solely from a set of view class definitions—much as the Java compiler operates upon application source code to produce byte-code for the Java virtual machine—doing so would prevent us from taking advantage of the wealth of information available within the application's runtime context. The same contextual advantage is sought by the just-in-time compiler frequently deployed within the Java runtime, however in our case we are also seeking sufficient context to implement a specific fragment of the process as a completely *statically scheduled pipeline*.

A particular view *instance* will often have constant bindings to other view objects and values supplied through its constructor, which are simply impossible to infer from the static context of the class alone. These bindings are crucial to our aim, since we achieve static scheduling by fully inlining the implementation of the top-level view's `draw` method. With instance bindings available within our code generation context, and thus concrete classes immediately resolvable, it is possible to fully inline the necessary method invocations, reducing the implementation to constant time operations acting directly upon the canvas abstraction.

Hence we invoke the code generators for our OPP within the runtime context of the application, and furthermore within the encapsulated context of the object instance, both characteristic

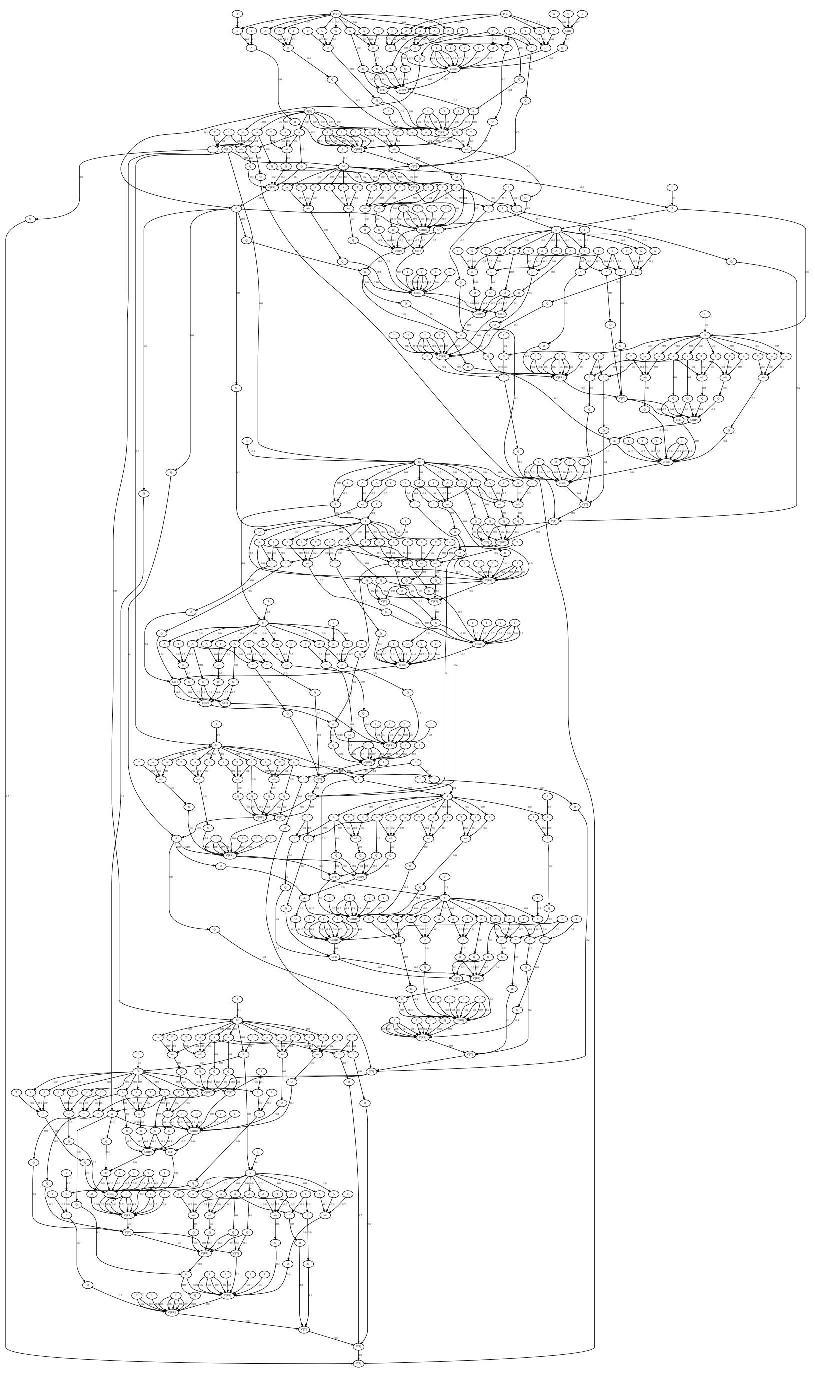
Figure 4.1: Schematic of the generated rasterisation pipeline. The dense clusters of closely connected nodes represent components within the pipeline and their associated inputs, derived either from constants or registers which are synchronised with the application state. The long edges are primarily queues for propagating values calculated earlier in the pipeline to later nodes, including clipping information. Nodes perform a variety of different operations including: (i) providing constant or mutable data values; (ii) computing fixed-point arithmetic; (iii) computing boolean values; (iv) transforming pixel coordinates; and (v) clipping and shading pixels.

elements of late-bound code generation. There are three senses in which this code generation process is late bound: first the delegate generators are polymorphically bound via dynamic dispatch; secondly the context within which the code generator is invoked is under control of the source program; and finally the bindings of that application context are available to the code generator itself. Importantly this implies that, by delaying code generation to an appropriate point in the process evolution, the presence of certain bindings within the runtime context may be ensured.

To illustrate the process of generating an OPP from a user interface in detail, we will first describe the translation of a trivial example: an application presenting just two immutable rectangles. Subsequently we will return to the example application of our experiment, and describe how the additional complexities of this more realistic example are handled.

4.7.1 Minimal example

Consider the following trivial view definition:

```
class TrivialView(View):
  def on_draw(self, canvas):
    return canvas.fill_rect(10, 10, 10, 10, 0, 255, 255) \
                 .fill_rect(10, 50, 20, 20, 255, 0, 255)
```

We will describe how an OPP for this view is generated by means of the `translate` operator which we introduced in Chapter 2. We have deliberately simplified various aspects of this process in order to avoid distraction away from the essential details. A full presentation is made in Section 4.8.

Our trivial user interface consists of just one view object, simply presenting a pair of filled rectangles, one cyan and the other magenta. We have defined this view class in Python, despite the fact that Python is not officially supported on Android, since it serves as our primary language for performing late-bound code generation, for reasons justified in Section 4.10.2.

In addition to diverging from the Android standard with respect to language, our API also differs slightly, instead adopting a functional interface for draw methods on view objects, and similarly methods on the canvas. Furthermore we have provided various methods for filling regions without explicit "paint" objects. Whilst not strictly necessary for OPP generation, these departures were taken to simplify both exposition and implementation. We discuss the rationale and implications of this decisions in greater detail in Section 4.10.2.

Strictly speaking, views on Android are drawn by invoking their `draw` method, and not `on_draw`, which is protected. This is because draw dispatches to child views of the instance, amongst several other responsibilities. In this minimal example there is no observable difference between the two methods.

Hence, in order to generate an OPP for the rasterisation of `TrivialView` we must translate the behaviour of `on_draw` to our OPP intermediate representation, for subsequent synthesis of that RTL-level description into the binary format accepted by our FPGA. To translate the behaviour of this method we use the `translate` operator, to perform late-bound code generation of the OPP:

```
mandate = None
top_level_view = TrivialView()
initial = BlankPixelSizedApertureCanvasNode()
pipeline = translate(
  OPP,
  immediate(OPP, TrivialView.on_draw, mandate),
  [ immediate(OPP, top_level_view, mandate), OPP([initial], [], initial) ],
  mandate
)
```

We have supplied `None` for the mandate since it is not used in this trivial translation. The result of the translation "`pipeline`" is an instance of the `OPP` class, an RTL representation of the behaviour of the method `TrivialView.on_draw`. Let us break down all of the steps performed during this translation, one by one.

1. An instance of `BlankPixelSizedApertureCanvasNode` is created to serve as the initial node of the OPP. It is a node that acts as a variable within the OPP, in the same way that an unbound name "`x`" would if we were targeting our translation to Python. In contrast it *is* annotated with a concrete type `PixelSizedApertureCanvas`, hence this node is type-restricted to represent an aperture to the notional canvas upon which the application draws, measuring just 1 pixel wide and 1 pixel high. The idea is to translate the process of drawing just one pixel of the canvas, leaving the horizontal and vertical position of that aperture unbound. Within the FPGA, since the OPP is pipelined by construction, we render the full canvas by streaming the list of pixel coordinates though the OPP in raster order, emitting one pixel per clock cycle from the end of the pipeline.

2. We represent the method `on_draw` on the class `TrivialView` as a term in `OPP` using the `immediate` function. This is necessary, since the translate operator assembles fragments only in terms of the target language.

3. An `OPP` term is constructed containing our instance of `BlankPixelSizedApertureCanvasNode`, the second argument provider for `on_draw` providing the argument canvas. This serves as the initial pipeline instance from which the eventual pipeline is grown by composing further nodes onto the end of the pipeline (instances of `OPP` are immutable). The first argument to the `OPP` constructor is a list of nodes within the graph; the second a list of connections describing which node outputs flow into which node inputs; and finally the third argument denotes the output node of the pipeline. Pipelines have exactly one output node, and all nodes have exactly one output connection also, hence when composing pipelines there is no ambiguity when combining the output of one pipeline with the other: implementing composition operators for pipelines is thus straightforward.

4. The `translate` operator is invoked, passing `OPP` as the target language, along with terms in `OPP` representing the method whose behaviour is to be translated and its two argument providers we have just constructed.

5. After checking various preconditions on its arguments, such as that all given terms are terms in OPP, translate enumerates all candidate code generators applicable to the provided function, and delegates to the most specific generator, the ordering of which we detail in Chapter 5. There are two possible code generator candidates for this example: a code generator attached directly to TrivialView.on_draw; or a generic code generator attached to the OPP class, capable of generating OPP terms from any Python function, by means of reification. The former takes priority if it exists, whereas the latter saves the programmer from having to explicitly define code generators for each of their functions. The ability to attach code generators directly to functions is primarily intended to allow for customisation of the code generation process, but since it is far simpler to describe this particular branch of the translation process, we assume that the following code generator has been explicitly attached to TrivialView.on_draw before the invocation of translate:

```
def trivial_view_on_draw_to_OPP(_, argument_providers, mandate):
  canvas_provider = argument_providers[1]
  with_rect = translate(
    OPP,
    immediate(OPP,
      canvas_provider.result_node()['output_class'].fill_rect,
      mandate
    ),
    [ canvas_provider, immediate(OPP, 10, mandate),
      immediate(OPP, 10, mandate), immediate(OPP, 10, mandate),
      immediate(OPP, 10, mandate), immediate(OPP, 0, mandate),
      immediate(OPP, 255, mandate), immediate(OPP, 255, mandate) ],
    mandate
  )
  return translate(
    OPP,
    immediate(OPP,
      with_rect.result_node()['output_class'].fill_rect,
      mandate
    ),
    [ with_rect, immediate(OPP, 10, mandate),
      immediate(OPP, 50, mandate), immediate(OPP, 20, mandate),
      immediate(OPP, 20, mandate), immediate(OPP, 255, mandate),
      immediate(OPP, 0, mandate), immediate(OPP, 255, mandate) ],
    mandate
  )
```

This laboriously written code generator might be associated with the `TrivialView.on_draw` method by assigning it to the "well-known" attribute `_translate_invoke_to_OPP` of the method object:

```
setattr(TrivialView.on_draw,
    '_translate_invoke_to_OPP',
    trivial_view_on_draw_to_OPP
)
```

The presence of this attribute is tested by `translate` when enumerating the candidate code generators for `TrivialView.on_draw`.

6. Next `translate` delegates to our code generator, forwarding the argument providers and mandate. Therefore `canvas_provider` is bound to `OPP([initial], [], initial)`. As is typical when implementing late-bound code generators, sub-tasks within the translation are further dispatched via recursive calls to `translate`. In this case we rely on obtaining an immediate reference to `fill_rect` in order to translate its behaviour, although in more complex scenarios the delegate method might not have been (fully) instantiated. We cannot statically assume the concrete class of the provided canvas, namely `PixelSizedApertureCanvas`, but instead resolve it at code generation time by inspecting the meta-data attribute '`output_class`' on the result node of the canvas provider. Since all of the arguments to `fill_rect` are constants, providers for each are constructed

by means of the `immediate` function.

7. The code generator for `fill_rect` is in turn invoked, and returns the pipeline `with_rect`, comprising the `initial` node and a node for rasterising the rectangle, specialised to the provided arguments. This code generator for `fill_rect` is attached directly to the `PixelSizedApertureCanvas` class. A practical implementation of our OPP generation system within Android would likely involve several parties, each delivering different software components. Since the canvas abstraction is generic to all Android applications we imagine that for practical deployments of the system described, code generators for canvas operations would be supplied by a small number of vendors for use by the larger set of application developers. Hence an application developer might reasonably take the availability of the `fill_rect` code generator for granted. Nevertheless its implementation might be given by the following method:

```
def aperture_canvas_fill_rect_to_OPP(_, argument_providers, mandate):
  queues = equalise_latency(argument_providers)
  rasteriser = FillRectNode(PixelSizedApertureCanvas)
  translation = concatenation(queues).concatenate(
    OPP(
      [rasteriser],
      [Connection(queues[port], rasteriser, port) for port in range(0, 8)],
      rasteriser
    )
  )
  return translation
```

We have elided the logic for clipping the pixel stream from this implementation, for simplicity—in the following Section 4.8.4 we describe how clipping is handled in general. This code generator simply generates a circuit which appends a `FillRectNode` to the pipeline. `FillRectNodes`, as we saw in Section 4.5 previously, take eight inputs and produce one output, namely the resulting pixel stream. Along with the input pixel stream the other seven inputs comprise the coordinates and fill properties of the rectangle. Each of the eight inputs are connected to the `FillRectNode` by the connection objects instantiated in the list comprehension: here we take advantage of the fact that by design the order of arguments to the canvas API match the order of inputs to the `OPP` node. The appended `FillRectNode` is annotated as the output node of the returned `OPP` by passing it as the third argument to the `OPP` constructor. The only subtlety within this code generator is found in the call to `equalise_latency`. Here we insert fixed latency queues to delay the arrival of arguments values, such that each arrives at the same time. The function `equalise_latency` performs this task, taking a list of `OPP` instances against which it interrogates each members' latency, returning a list of similar `OPP` instances each delayed to present equivalent latency. It is worth noting that we represent all circuitry for delivering arguments as `OPP` instances, whether they are canvas arguments, numeric arguments, or any other kind of value: a property that we rely upon heavily during the pipeline generation process.

8. This intermediate pipeline is composed similarly with the node for rasterising the second rectangle, to obtain the final pipeline.

The resulting OPP is bound to `pipeline`, and consists of two rendering nodes connected in series, with a miscellany of supporting nodes delivering various constants to the pipeline:

Subsequent synthesis is performed by casting this pipeline to Verilog, which just discards various metadata held by our intermediate representation, similar to the phenomenon of "type erasure" during the compilation of Java programs. When implementing the pipeline for a specific FPGA, each vendor provides their own tool-chain, typically accepting Verilog as a source language.

Depending on the nature of the hardware target, various signals exposed by our pipeline must be linked with the requisite hardware: at the very least the global clock and the pixel output stream. Such linkage declarations are handled in either an ad hoc manner for each target, or alternatively according to a standardised API for integrating rasterisation pipelines into a hardware system. We do not attempt to define the latter within this thesis.

4.8 Practical pipelines

Whilst generating the pipeline for our trivial example faithfully illustrated the essential architecture of our code generation system, more realistic applications typically involve considerable additional code generation complexity. These complicating factors must be properly accommodated in order to generate an OPP specialised to any non-trivial application. Those factors which we shall address in detail within this section include:

- Use of constancy and bounding annotations to restrict the variability of the canvas,
- Preemption and synchronisation of application state with the pipeline,
- Code generation by means of reification,
- Coordinate transformations and clipping of rasterised regions.

Finding a solution for each of these issues was necessary in order to correctly generate a pipeline for the view synthesised in our experiment detailed in Section 4.6. Here we take each factor in turn, and describe how it was solved within the context of our application.

4.8.1 Employment of annotations

Whilst there often exists a statically scheduled OPP for rendering the user interface for many applications, that pipeline is not always tractably inferred from its formal representation. Typically the formal description permits a greater degree of variability in the user interface than is either necessary or utilised: variables are often assigned only once yet not declared constant, and lists are left unbounded yet only ever populated with at most half a dozen elements. Generating an OPP, for execution on an FPGA, with respect to an under-constrained user interface is either impractically wasteful of the FPGAs resources, or worse, infeasible to statically schedule. In Section 4.6.1 we defined a set of annotations to allow the application developer to formally declare certain kinds of constraint, respected by their application's user interface. Where these constraints bear upon the tractability and practicality of the OPP generation, a mechanism must be available by which those relevant annotations can influence the generation of the pipeline. There is considerable design freedom in devising this mechanism, which we broadly refer to as *annotation steering*.

In our example we chose to implement a mechanism which extends the "view inflation" process, already native to Android applications. As we have already seen, Android applications define a blueprint for their application's views in XML, from which the Android system automatically instantiates view objects: this process is known as *inflation*. Many view objects may be instantiated from the same XML blueprint.

To convey any annotations applied to the blueprint, we opted to insert a second phase into the view inflation process. Firstly each XML blueprint is inflated into a runtime class, subclassing its nominal view class but adorned with overriding members, methods and code generators which respect the annotated constraints. Secondly each view instance is instantiated from this dynamically generated subclass.

This two-phase inflation scheme simplifies the process of translating non-immediate views. Even though a particular view instance might not be available at translation time, at least the constraints necessary to generate its OPP are available via bindings on its concrete class. In short, we use dynamic class generation as a technique for binding-time separation in the OPP compiler.

Consider for example the following XML blueprint, for a recycler view:

```xml
<RecyclerView
        ID="MainRecycler"
        Left="32" Top="198" Right="608" Bottom="1086" Background="#FFFFFF"
        FixedBackground="true"
        UniformChildView="ScheduleItemView" MaximumChildCount="4">
</RecyclerView>
```

This blueprint is adorned with three annotations: `FixedBackground`, `UniformChildView` and `MaximumChildCount`. During phase one of view inflation we create a runtime subclass of

`RecyclerView`, extended with an implementation of the static method `is_background_colour_fixed` which returns `True`, and `fixed_background_colour` which returns the colour `#FFFFFF`. This is sufficient to support efficient OPP generation, since the default code generator on the `View` parent class was defined to respect the results of these static methods. Subsequently this dynamic class is further subclassed in order to incorporate the constraints on its children. In particular we generate a subclass which instantiates the list holding all of the recycler view's children from a specialised list class. This specialised list class is also dynamically generated in a similar fashion: a class is generated at runtime which implements the generic list interface, but has static bindings for the maximum number and concrete type of its members. Attempting to add members to lists instantiated from this specialised class, not adhering to these two constraints, will raise a fatal exception.

Consequently code generators for methods of this list are defined, which annotate the `'output_class'` of their result node with the given statically bound concrete class, in this case `ScheduleItemView`. Hence the inflated recycler view, and its list of children, need not be instantiated at OPP generation time for the pipeline to be efficiently generated, since the constraints are all bound statically against the class structure, albeit at runtime.

It is precisely in this manner that all type information is propagated from annotations, and between code generators, at translation time.

4.8.2 Preemption and synchronisation of application state

The user interface of our trivial example was constant—it did not evolve its appearance over time or due to interactions from the application's user. This is rather contrived, since most application user interfaces evolve as the state of the application changes, as does the user interface of our experimental application proper.

Since the generated OPP is intended to serve over the entire lifetime of the view object, it must necessarily contain mutable variables which are updated when the state of the application changes. It would be rather expensive to synchronise the state of all the application's variables with the OPP, since most do not directly impact on the appearance of the user interface. Instead we describe a mechanism by which the state synchronisation interface may be delineated automatically during OPP generation. This involves a novel abstraction known as the *preemption set*, introduced previously in Section 4.6.2.

In general we populate the preemption set whenever a mutable object is represented in the OPP via application of the `immediate` function: capturing the fact that the state of this object must be synchronised with the OPP. For example if a code generator represents some mutable object `foo` within the OPP by calling `immediate(OPP, foo, mandate)`, then `foo` is added to the preemption set attached to the mandate: `mandate.preemption_set.include(foo)`. This can be done either explicitly, or automatically provided that all candidate objects consistently annotate whether they are mutable or immutable (inferred mutability is a conservative default).

In general it is possible for the OPP to depend on mutable objects which are not yet instantiated at translation time. This scenario occurs within our experiment, whereby view instances which are dynamically added and removed from the recycler view during scrolling are not immediate during OPP generation. Whilst this scenario is complicated to handle in general, the hierarchical structure of views in Android presents a simple solution.

The solution is thus: despite the fact that some views are not immediate, since we perform OPP generation at the point the top-level view instance is bound, then at least one ancestor of all views in the user interface is certainly immediate. This immediate ancestor view is added to the preemption set, and thus represents the state of both itself and all of its *potential* children. In our experiment the recycler view itself was immediate, and so during OPP generation the recycler view instance was added to the preemption set. By virtue of the view hierarchy, the recycler view can be unambiguously considered to own its children, and so the variability of its child views are notionally factored into its own state.

Preemption example

Consider again our trivial example, but with a modification that the x-coordinate of the filled rectangles depend upon an attribute of the `TrivialView` instance:

```
class TrivialView(View):
  def on_draw(self, canvas):
    return canvas.fill_rect(self.offset, 10, 10, 10, 0, 255, 255) \
                 .fill_rect(self.offset, 50, 20, 20, 255, 0, 255)
```

Since the object is mutable, it must be added to the preemption set at the point it is represented in the pipeline. This requires overriding the default behaviour of `immediate`, since by default it merely represents objects in the target language by reference, resulting in non-synthesisable nodes within the OPP intermediate representation (note that since the state of immutable objects are always inlined, a non-synthesisable node is elided in those cases). We employed the following override to `immediate`, for all mutable objects involved in our experiment:

```
def represent_mutable_object_in_OPP(self, mandate):
  return mandate.preemption_set.include(self)
```

This override is attached to the class via an attribute `_represent_in_OPP`, which is delegated to preferentially when calling `immediate` against the `OPP` target:

```
setattr(TrivialView, '_represent_in_OPP', represent_mutable_object_in_OPP)
```

The instance of `TrivialView` which is bound to `self` is thus added to the mandate's preemption set. Including an object in the preemption set generates an `OPP` object which acts as a provider for the included object. To generate this provider, the preemption set delegates to the class of the included object, calling its `preemption_interface` method. In our experiment this method simply returns a single node pipeline fragment, which delivers a registered bit-string, and is annotated as providing an object of the given class. Each class can override this method for serialising its own state to a bit-string: the only constraint when targeting `OPP` is that the (maximum) length of the bit-string must be defined at translation time.

The salient point here is that the preemption set maintains a link between the included object and the node representing that object within the pipeline. By iterating though the members of the preemption set, serialising their state and assigning the resulting bit-strings into the registers of the associated nodes in the pipeline, allows the state of applications to be synchronised with the pipeline periodically.

As we discussed in Section 4.6.2, Android development “best practice” states that any mutable application state which interacts directly with the rasterisation process should not be modified whilst the user interface is being drawn, since doing so could lead to visual corruption of frames. Consequently, in our experiment we relied upon this property to schedule our synchronisation with pipeline: a snapshot of the state of all objects in the preemption set is taken just before each frame is rasterised, a binary comparison of the serialised state is taken against the previous frame’s snapshot, and any changes are sent to the FPGA, so synchronisation occurs at most once per frame.

Whilst we might simply assume that the states of objects in the preemption set are not mutated during rasterisation, given that we are translating the rasterisation process, this provides a convenient opportunity to verify this assumption, at least with respect to some other more basic assumptions. If we assume that the runtime objects which represent the user interface, including all views and their dependent objects, are only mutated from the main Android thread responsible for rasterisation, then we can conservatively verify the absence of state mutation during rasterisation, by raising a *translation exception* if any operation entailed by calling `draw` on the top-level view cannot be determined to be free of mutating effects. Since the analysis required to perform this verification is very similar to the analysis required for the translation itself, we interwove the verification and translation processes together. This is only possible since we perform full inlining of all methods called within the rasterisation process, and so can easily check each operation involved is free of mutating side-effects.

4.8.3 Code generation by means of reification

In previous sections we described the process of OPP generation using explicit code generators, attached manually to every function or method involved in rasterising the user interface. Whilst it might be necessary to write explicit code generators for some finite set of primitive operations, an OPP for any user interface built using those primitives could in theory be obtained by composition of those primitive generators. To expect an application developer to manually write a code generator for each method defined within their application would largely defeat the purpose of code generation—they might instead more simply define the OPP explicitly, rather than generatively. One portion of the OPP generated in our experiment was formed by means of reification, specifically the "floating action button", shown in Figure 4.2.

When an application developer defines their user interface in Android they do so using imperative code, at least in the most general case. Hence we must be able to compose OPP fragments corresponding to those primitive operations on canvas objects, by inspecting the source definitions and runtime environment of the application.

In short, we must allow `translate` to reify aspects of the source function, and use these reified forms to compose the method's translation without recourse to an explicit code generator for every function. This better aligns with what might be expected of an operator called `translate`—an operator which takes terms in some source language and emits equivalent terms in some target language. However, `translate` remains an operator which acts upon *function providers*—target terms which evaluate a function *closure*—and thus provide access in general to not only the function's source syntax, but also the scope over which the syntax is closed, in addition to other miscellaneous details of its runtime context.

Chapter 5 describes in detail how `translate` performs translation by means of reification. Summarily, the syntax of the function body, or method body, is reified to a term accepted by some meta-circular interpreter. Since evaluation of that term by the meta-circular interpreter would effect invocation of the function, `translate` simply recurses, forming its result by translating the *evaluation function* of the interpreter given the reified syntax of the function and an appropriate *scope provider*. Infinite regress is avoided by attaching an explicit code generator to the *interpreter*, which thus serves as a single generic generator for any reifiable function.

The only substantial subtlety here has to do with provision for the scope. Whilst the syntax of any immediate function is necessarily immediate, the same cannot be said for the scope within which that syntax is ultimately evaluated, since certain function arguments might be unbound at translation time. Furthermore if that scope is mutable, it must be instantiated separately for each function invocation.

Instead a term is formed which nominally instantiates the scope instance at runtime, analogous to the instruction sequence for a typical CPU which manipulates the stack pointer to establish a new stack frame. Various metadata are attached to the provider, recording any information which is immediately available that pertains to the provided scope, for instance the closest immediate ancestor of the scope, if it is known. In the context of our experiment, all translated functions happened to be immediate, so their closest immediate scope ancestor was the lexical scope of their definition.

Provision for any argument bindings was accomplished by means of contextual generators, which we described later in Section 5.6.4. Briefly, contextual generators are code generators attached to a provider, which generate terms valid within the same context as the provider, but not necessarily within a wider context. As such a contextual generator for each argument was constructed, such that resolution of that name at translation time would return the fragment of the OPP providing that value directly, rather than via runtime lookup, which is not possible to schedule statically, nor would it be particularly efficient. These contextual generators and immediate ancestor scopes are used preferentially by all delegate code generators, when translating the evaluation of names within the function's syntax.

In our experiment the scope providers formed in this manner were *non-synthesisable*, but given that in all cases bindings accessed via the scope were resolved either to constants through an immediate ancestor, or arguments via a contextual generator, then all scope providers within the OPP were ultimately left disconnected from the pipeline and safely elided before casting to Verilog. Had this not been the case then the state of certain mutable bindings would have needed synchronisation with the pipeline in a manner described in Section 4.6.2, or worse resulted in failure to generate a statically scheduled pipeline.

In general one might attempt to translate the behaviour of non-immediate functions to `OPP` terms, for example functions for which the syntax is immediate but the lexical scope closing the function definition is not, or possibly even cases in which the syntax is not fully immediate. Such translation challenges were not encountered within this experiment, and are outside the scope of this chapter.

Having outlined the procedure by which we generate code by means of reification, it is perhaps instructive to compare our approach to the well known technique of specialising an interpreter to achieve compilation, since there are superficial similarities. Specialising (partially evaluating) an interpreter to a given source program yields a translation of the source program in the implementation language of that interpreter—this is known as the *first Futamura projection* [Fut99]. Although related, this is quite distinct from the approach taken here. Whilst we do apply the `translate` operator to the evaluation function of an interpreter, that of our application's source language, the language of the resulting translation bears no relation to the implementation language of the interpreter. Furthermore the first Futamura projection implies that the translation is generated by means of reifying the implementation of the interpreter: from our perspective this is an implementation detail—it is not important whether or not the interpreter is adorned with explicit code generators, relies on code generators supplied by its own implementation language, or perhaps makes use of even further indirection. Since within our experiment we wrote an explicit code generator for the evaluation function of our metacircular-interpreter, which itself was not implemented in `OPP`, the approach we took in this chapter is more closely related to conventional compilation than to compilation by means of the first Futamura projection.

4.8.4 Coordinate transformations and clipping of rasterised regions

The operations supported by Android's canvas API extend beyond those for simply drawing geometric primitives on to the canvas. Notably there are also operations for applying *coordinate* *transformations* and setting *clipping regions*. Both of these kinds of operations were necessarily used in our experiment.

Typically when a child of a recycler view is drawn, the method of the child view which draws the child's content does not explicitly perform translation and clipping of its own drawing commands. The content of the child is drawn in a relative coordinate system, in which the child view is aligned to the origin of its canvas. The drawing commands are issued in this relative coordinate system, and are applied to the underlying canvas, which is shared by various views, under a coordinate transformation matrix. The transformation matrix is set up by the parent view when delegating drawing operations to any of its children (in a recursive manner) based upon the relative position of the child within the parent. Similarly a clipping region is established to prevent drawing operations performed by the child from affecting regions of the canvas outside the bounds assigned to the child. As such, each child of a recycler view simply draws itself in full at the origin of the canvas supplied, leaving it to the recycler view's transformation matrices and clipping regions to ensure that the correct subregion of the view is rasterised at the appropriate offset, and within the bounds of the recycler view.

In order to implement these necessary transformation and clipping operations within our OPP we devised a couple of techniques, both of which introduce additional nodes into the pipeline, and so materially increase its ultimate latency.

Transformation nodes

Transformation operations applied to the canvas are directly translated into transformation nodes within the pipeline. Since the coordinates of each pixel are part of the datapath in our OPP architecture, transformations to the canvas' coordinates are affected by appending nodes to the pipeline which consume and emit pixels unchanged, except for their horizontal and vertical coordinates, which are mapped into the relative coordinate system. Android permits arbitrary linear transformations to the canvas in general, which require a matrix multiplication to be performed within the pipeline. However, in many cases the transformation may be determined during OPP generation to be a pure geometric translation, in which case a simpler specialised node consisting solely of adders is appended preferentially.

When delegating translation to a child view, the code generator for the parent must insert transformation nodes into the pipeline before and after the fragment returned by the delegate, in order to both map the pixel stream into the coordinate system of the child, and map back again.

Clipping nodes and the window stack

Unlike canvas transformations, clipping operations cannot be implemented in the pipeline using a pair of opposing operations which act solely on the OPP datapath. Android manages restoration of the clipping region in unison with restoration of the transformation matrix, using a stack maintained within the canvas instance. We refer to this stack as the *window stack*. Frames can be pushed and popped from the window stack in order to save and restore the state of the coordinate system and current clipping region.

In our experiment we found it sufficient to represent this stack implicitly within the pipeline, using metadata attached to each OPP node, rather than explicitly within the runtime state of the pipeline. Since we adapted the canvas API to a functional interface—the implications of which are discussed further in Section 4.10.2—the state of the window stack is simply an immutable value associated with the canvas instance. All canvas operations which act upon the window stack push or pop an immediate number of frames, and so the stack depth may be maintained as a metadata attribute on each node in the pipeline, provided that stack manipulating operations are applied unconditionally, which was indeed the case in our experiment. The frames of the stack themselves are not immediate, since the corresponding transformation and clipping regions are data dependent. Hence the frames of the stack are represented using OPP fragments, which provide the requisite values.

We decided to represent clipping regions using a binary canvas, itself generated by an OPP fragment. Consequently in the presence of clipping, the resulting pipeline consists of several fragmentary pipelines which are joined together at clipping nodes. Since our OPP representation is simply an acyclic graph of nodes, this is straightforward to represent, and naturally supports the representation of arbitrarily shaped clipping regions.

As the pipeline is constructed, the window stack metadata property is populated with frames consisting of pipeline fragments, which provide the clipping and transformation parameters. These pipelines, which are initially just metadata, do not form part of the `OPP` graph until the window stack is popped. Popping a frame from the window stack links these pipeline fragments into the graph: transformation nodes and clipping nodes are appended to the result node of the pipeline, taking their various inputs from the pipeline fragments which were popped from the window stack.

When this linkage occurs, as with all cases of pipeline composition, care must be taken to ensure that values arriving at different input ports of the same node have equal latency. Since the point at which clipping parameters are computed within the pipeline often differs substantially from the point at which they are used—typically the latency of computing the complete nest of sub-views intervenes—clipping can contribute a considerable number and depth of queue nodes to the generated OPP. In Section 4.10.1 we discuss the degree to which is issue might be mitigated, and to what extent it is necessary given our imposition of a static schedule upon the pipeline.

4.9 Results

The rendering pipeline for our experimental scenario was generated, synthesised, and placed & routed for a Microsemi PolarFire MPF100T device, in order to quantify its utilisation. The vendor's power estimation tool [MTIMPP21] was then used to calculate the power consumed by our pipeline under a range of different loads, from static image generation to a complete rasterisation every frame.

Type	Used	Available
Global clock networks	1	24
D-type flip-flops	13260	108600
4-input LUTs	14594	108600
SRAM blocks	245	1008
DSP blocks	11	336
IO pins	72	318

Table 4.1: Pipeline resource utilisation summary. Only one global clock network is used, since every node of the pipeline shares a common clock. D-type flip-flops are typically used to register the results of nodes, and their utilisation is hence closely correlated with the number of look-up tables (LUTs) used. All of the combinational logic for the pipeline is implemented using 4-input LUTs. SRAM blocks are typically used to implement the FIFOs for synchronising signals at node input ports. IO pins are allocated for both the inputs and outputs of the pipeline, in addition to its state synchronisation interface.

4.9.1 Device utilisation

A schematic of the experimental pipeline is shown in Figure 4.1 and its device utilisation, broken-down by resource type, summarised in Table 4.1. All of the nodes within the OPP are clocked by the same global clock, and the number of SRAM blocks largely depend on the number of synchronisation queues within the pipeline. The longest of these queues can be seen clearly within 4.1, corresponding to the long vertical edges uninterrupted by nodes. These queues are employed to delay the use of values which are computed at early stages of pipeline, and consumed both at early and late stages of the pipeline—typically such values are windowing and clipping parameters, the use of which 'surrounds' the windowed or clipped sub-image.

The majority of the computation is performed within the FPGA's 4-input lookup tables (LUT), with a minority of operations taking advantage of the digital signal processing (DSP) blocks for integer multiplication. The heavy use of D-type flip-flops is due to the latching of results between each node of the pipeline, in an attempt to reduce the circuit's maximum combinatorial logic delay, therefore enabling a higher clock frequency. This is in turn motivated by the observation that the faster the circuit is clocked, the greater the percentage of time which can be spent in an inactive power state having lower static power consumption.

The number of IO pins exceeds the minimal output number of 24, for delivering one 24-bit pixel value per clock cycle, since the pipeline must be fed with per-cycle x and y raster coordinates,

Parameter	**Value**
FPGA product range	Microsemi PolarFire
FPGA part code	MPF100T
Pipeline depth	21 visible objects
Rasterisation resolution	1156 pixels × 640 pixels
Rasterisation refresh rate	60 frames / second
Colour depth	24-bit (RGB888)
Target clock frequency	225 MHz
Target maximum active duty cycle	20% (otherwise in Flash*Freeze state)
Flash*Freeze state power consumption	15 mW
Core voltage	1.0 V
Junction temperature	25 °C

Table 4.2: Device and experimental parameter summary. The user interface of our experiment consists of 21 visible objects, rasterised at a maximum frame rate of 60 Hz to an 18 Mbit canvas, resulting in a total output bandwidth of just over 1 Gbit / second. If the FPGA is clocked at our target frequency of 225 MHz then for the majority of the time (80%) it remains idle, transitioning in and out of a low-power (Flash*Freeze) state. Nominal voltage and temperature parameters are assumed.

and also periodically synchronised with the state of the application. It is clear from these utilisation numbers that a more complicated user interface could potentially be rasterised within this device: when utilising a larger fraction of the device the percentage of power consumed due to static leakage would therefore be lower.

4.9.2 Device and experimental parameters

Table 4.2 summarises the various device and experimental parameters, with respect to which we estimated the power consumed by the pipeline. The pipeline contains nodes for rendering 21 visible objects of fixed type, specialised to the applications. Whilst objects might in general correspond to shaded ellipses, rounded rectangles or even blocks of text, in our experiment all objects were rectangles, either flat shaded or shaded with a repeating two-colour sequence. A screenshot of this experimental application is shown in Figure 4.2. Given the inherent utilisation cost of the pipeline infrastructure, the incremental cost of substituting more complicated geometry or shading patterns is not necessarily overwhelming.

The raster resolution, colour depth and refresh rate were chosen to represent the *minimum* specification necessary to deliver a contemporary smartphone user experience. The device's

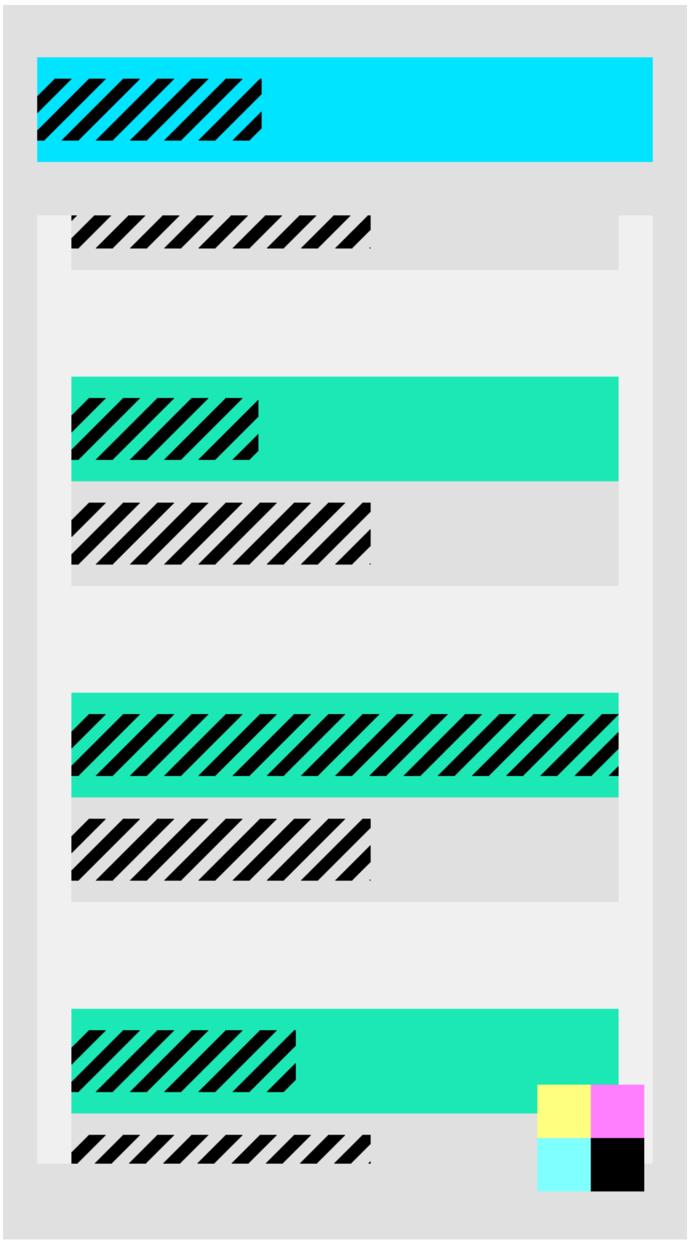

Figure 4.2: Screenshot of our experimental application. The various coloured regions correspond to different views and sub-views within the user interface. Regions shaded with black diagonal lines correspond to text views, which are not rasterised within our pipeline, and require external compositing. Although this user interface is extremely simple, observe the central recycler view consisting of similar, but non-identical green entries, clipped within a rectangular bounding box. The contents of these green entries depend upon application data, reflected by the differing widths of the text views. The four squares at the bottom right represent the operating system's "floating-action button", which partially overlays the recycler view. The recycler view depends not only upon application data for its text content, but also for its scroll position: all such mutable state is represented and reflected within the rasterisation pipeline using externally accessible registers. This image was generated by executing the Verilog RTL representation of the pipeline within an FPGA simulation environment based upon Verilator [Ver21].

Type	**Value** / mW
Static overhead	43 (16%)
Clock network	30 (11%)
Logic blocks	70 (26%)
DSP blocks	9 (3%)
SRAM blocks	34 (13%)
IO switching	50 (19%)
IO current	32 (12%)
Total	268

Table 4.3: Device power consumption breakdown (in active state). Nearly half (47%) of the total 268 mW consumed by the pipeline is due to static leakage currents and IO signalling—fixed properties of the FPGA device, and the pipeline interface respectively. A further 29% of the power consumption can be attributed to computation, specifically within the FPGA logic and DSP blocks. The remaining 24% is dissipated within the clock distribution network (control signalling) and SRAM queues (synchronisation).

Flash*Freeze state consumes 15 mW of power, and provides a lower-bound on the power consumed by our pipeline, since we do not consider fully powering down the device between frames. In order to minimise the mean power consumed by the pipeline, the amount of time the FPGA spends in this state must be maximised. Within this low-power state the device can perform no useful computation, but simply retains any state necessary to return to an active state when a frame must be redrawn.

The core voltage of 1.0 V is standard for the device, and the junction temperature for our power model assumes the device is not subject to considerable self-heating. The temperature in a real device would depend heavily on the thermal dissipation system surrounding it, and also the ambient temperature of its environment.

4.9.3 Estimated power consumption

A breakdown of the power consumed by the pipeline *whilst active*, estimated using the FPGA vendor's power estimation tool [MTIMPP21], is given in Table 4.3.

We observe that 47% of the total 268 mW consumed during rasterisation is due to a combination of static current leakage and operating the requisite input / output signalling (IO). Therefore almost half the power dissipated is not spent directly on computing the rasterised image itself, but in supporting the computation environment. Notably the 42 mW consumed due to static leakage is almost three times greater than the total power consumed within the Flash*Freeze state.

Approximately 29% of the total active power consumed is spent on core computation, namely within the FPGA's logic blocks and DSP blocks. The remaining 24% is spent on computational overhead, namely the clock network to synchronise the control path, and delaying values within SRAM to synchronise the data path.

If the device is called upon the render a full frame 60 times per second it would necessarily spend at most only 20% of the time in this active state—since a full frame can be delivered in about 3.3 ms given our resolution and clock frequency, with one pixel delivered per rising edge—spending the remaining 80% of the time in a low-power (Flash*Freeze) state consuming only 15 mW. This results in a maximum mean power consumption of $(0.2 \times 268 + 0.8 \times 15) = 65.6$ mW. Given that power consumption scales linearly with the mean percentage of frames being redrawn, the relationship between redraw rate and power consumption is given in Figure 4.3.

Assuming a user-interaction pattern representative of typical applications, where every frame does not require a full redraw, we conclude that our experimental application's rasterisation pipeline would consume approximately 30 mW during use, when executed within this MPF100T FPGA device.

4.9.4 Estimation accuracy

Subsequent timing analysis revealed that a minority of sub-circuits were insufficiently pipelined to meet timing at 225 MHz—a critical path within the rendering pipeline included too many gates between its bounding registers, such that the combinational logic would not reliably settle within one clock period at this target frequency. Additional pipeline stages would be needed in order to meet our timing objective, necessitating a refinement to the design of certain rendering components. It is not expected that these design modifications would *radically* affect the power values quoted above, but an increase in the order of 20% would not be surprising,

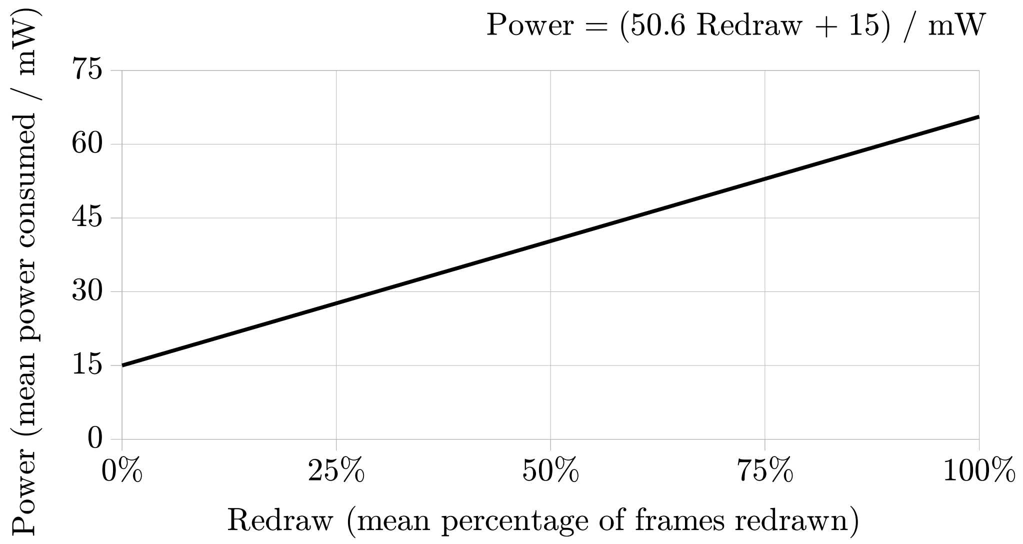

Figure 4.3: Power consumed by our experimental pipeline with respect to redraw rate. This graphical relationship was generated from estimated values, calculated using our FPGA vendor's power estimation tool [MTIMPP21]. The power estimator provides power consumption values corresponding to different FPGA states. The redraw rate corresponds to the duty cycle of a single active state, versus a single inactive state, giving rise to this linear relationship. The power consumed due to state switching has not been included, as it is assumed to be negligible.

and so an error of this magnitude is attributed to our estimate. Although the vendor's power estimator [MTIMPP21] did not provide explicit error bounds to the above values, we would similarly not be surprised to find that in practice these theoretical numbers differ from practical measurements by 10% or more. Combined we conclude that our power estimation error is at least 30%, and we would not be confident in claiming an accuracy of any better than 50%.

4.10 Evaluation

The aims of our experiment were twofold: (i) to demonstrate a candidate low-power rendering architecture for contemporary smartphones, and (ii) to generate instances of that architecture from smartphone applications defined in terms of contemporary abstractions. In this section we will evaluate the extent to which our objectives were met.

4.10.1 Low-power rendering architecture

As discussed in detail within Section 4.4, the object-processor architecture we selected for low-power implementation was justified on the basis of three principles: firstly that the pipeline processors could be specialised precisely to those required by the current application; secondly, due to the specialisation achieved, the resulting pipeline would admit a static schedule; and finally that the physical arrangement of the resulting pipeline would be amenable to place-and-route optimisation, in order to reduce the energy cost of of communication. Whilst our implementation faithfully adhered to each of these principles, it is unclear whether the resulting pipeline consumes less power than contemporary solutions, such as performing rasterisation within a GPU.

Static power

The availability of integrated circuits admitting place-and-route optimisation within their physical structure (spatial computing platforms) are fairly limited, at least for mobile computing, compared to the vast array of tradition fetch/execute computing solutions. FPGAs are currently the only widely available category of such device, although that may be set to change in light of recent developments in the space of AI accelerators [RMJ+21].

Since the granularity at which FPGAs may be programmed is at the level of individual bits, or small collections of bits, rather than bytes or larger words, the physical area of the FPGA which is given over to the gates and routing switches which support its reprogrammability is relatively large. For reasons described in Section 4.3.2 this gives rise to a considerable leakage current through the device, and consequently a larger static power consumption than would be necessary at a coarser level of granularity. Nevertheless since all spatial computing platforms necessarily distribute their programmability across the silicon chip, instead of restricting command words to a few select registers, there is likely an inherent increase in static power associated with this approach, at least when comparing implementations built using the same silicon process node.

In our experiment, despite using a Flash-based FPGA equipped with a power gate to its matrix, the average power dissipated due to leakage alone was at least 15 mW, within a typical average power consumption of 30 mW. Without context, it is difficult to properly evaluate this figure, but given the simplicity of the application rendered in our experiment, and the overall power budget of any handheld device limited to perhaps a few watts of dissipation, it is substantial enough to raise concerns about the feasibility of achieving low-power OPP rasterisation on this kind of spatial computing platform.

Power consumption in context

Unfortunately, we were unable to compare our power consumption results with those of a contemporary smartphone. In addition to such figures being commercially sensitive, with device vendors disinclined to make this information public, it could be rather difficult to make a fair comparison. Since we are particularly interested in static power contributions, it is unclear which current within any given complex device would be the right one to attribute *solely* to the rasterisation circuitry. Carroll and Heiser [CH10] did perform a comprehensive study of an Android smartphone commercially released in 2008, which was made possible by the public availability of the device's schematics. Despite this unusual degree of transparency with respect to the device's design, they still had to resort to some "combination of direct and subtractive measurements" to quantify the power consumed in the graphics subsystem.

In one of their benchmarks, of an email-centric use case, they estimated that approximately 80 mW of power in total was dissipated in the graphics circuity, a number which did not vary considerably across their other benchmarks, other than video playback. This perhaps suggests that the major part of that figure is accounted for by the static power consumption of the GPU, with the dynamic power accounting for the order of 10 mW variation between the scenarios they tested. This is highly speculative, and probably quite unlikely.

Furthermore it is unclear to what extent their figures reflect those of contemporary devices, with over a decade of research and development separating contemporary devices from their specimen. Finally the rendering complexity of our experiment is certainly significantly simpler than any of the use cases they investigated, so a direct comparison of our figures to theirs is not particularly informative.

Programming duration and memory fatigue

The use of Flash as a storage medium within an FPGA trades a reduction in static power for an increase in programming duration and the rate of device deterioration on reprogramming. The type of Flash memory adopted by Microsemi for the PolarFire MPF100T which we used in our experiment has a programming duration of several seconds [MTIMPP19]. This alone does not preclude dynamic reconfiguration of the device, since the CPU or GPU could be used to rasterise a user interface for the first few seconds of use, switching over to the FPGA once the programming is complete.

More pernicious is the fatigue suffered by the Flash memory when modified, since the particular technology used in the PolarFire MPF100T is only rated for approximately one thousand erase and write cycles [MTIMPP19]. This places a practical limit for reconfiguration of the FPGA to at most once per day for a typical smartphone's lifetime.

Consequently in order to use a PolarFire MPF100T for OPP rasterisation, either a predetermined set of pipelines would need to be selected and programmed into the FPGA, perhaps on each software update, or alternatively profiling the most frequently used views could allow for the set of programmed OPPs to adapt to usage patterns, with a daily cadence. Given this deployment strategy the energy cost of generating and optimising the OPP is amortised over at least one day. If a more dynamic strategy were to be adopted it would become increasingly important to also account for the energy spent on OPP generation, and within the FPGA compiler toolchain—something we did not attempt within our experiment.

Numeric representation

It was only possible to accommodate our pipeline within this FPGA device by adopting a fixed-point numeric representation. Had we selected an IEEE 754 floating-point representation instead, such as is widely used in CPUs and GPUs, the pipeline would have not only exceeded the space available in our experimental device, but also incurred a substantially higher power consumption, had a large enough FPGA been substituted. This is simply because the power consumed by an FPGA of the same type is always larger given a larger device, or a greater number of internal units toggling per clock transition. We found that a 16-bit fixed-point representation, with just 2 fractional bits, was sufficient for our experiment. FPGAs have the advantage that any number of bits can be used to represent numbers, and power and space are saved for each bit elided. This representation scheme was chosen manually, and no attempt was made to infer the minimum numeric representation automatically from the application. It is an open question whether this same representation would satisfy the requirements for other applications, or whether our approach would require the application developer to define the representation for each different user interface.

Queueing overhead

Our hypothesis, stated in Section 4.4, that the elimination of dynamic control-flow circuitry within the rasterisation pipeline would result in reduced power consumption, can now be examined. The argument used to justify this hypothesis supposed that if a static schedule could be found for the pipeline, then both the static and dynamic power which would otherwise be expended in the dynamic control-flow logic could be saved. Whilst our experimental OPP did admit a static schedule, the static schedule we implemented required several deep queues, in order to delay values arriving at nodes by the requisite number of cycles. After synthesis using the FPGA vendor's toolchain, a considerable fraction of the device's resources were allocated to queuing. Many of the longest queues, seen in Figure 4.1, were introduced as a consequence of view transformations and clipping. This is because intermediate values pertaining to the transformation and clipping of sub-views tend to straddle the portion of the pipeline responsible for rasterising the sub-view—for instance the same transformation coordinates must be applied before the sub-view fragment of the pipeline as subtracted afterwards. Those for the outermost sub-views of the user interface can become particularly long, approaching the total length of the pipeline itself.

In light of this observation we might re-examine the justification we give for our hypothesis. Since the imposition of a static schedule upon the pipeline in our experiment required a non-negligible amount of queueing circuitry, it is unclear whether the dynamic control-flow circuitry we avoided would have been less efficient. Naturally there are many dynamic control-flow implementations which would suffice for our application: in future work it would be interesting to select one reasonably efficient implementation and evaluate quantitatively how it compares. Doing so would help establish the degree to which dynamic control-flow is well motivated from an efficiency perspective.

The schedule we used in our experiment is certainly not the only static schedule admitted by the application, in certain scenarios nodes could be rearranged to displace latency around the pipeline, reducing the total proportion of the pipeline dedicated to queuing. Occasionally shared intermediate results might also be recalculated more economically than delaying the value calculated by earlier nodes. For these reasons it is possible that a lower power static schedule could still be found, perhaps by performing various such post-processing optimisations on the OPP representation.

Temporary storage

Historically, one of the primary advantages motivating research into object-oriented rasterisation, within which processor-per-primitive techniques are just one approach, was their economy with respect to temporary storage—in Section 4.11 we refer the reader to literature outlining the historical context, and certain notable architectures which have been presented. Having presented a method for object-processor pipeline generation for an Android application, it is worth considering if this historical advantage is still relevant to contemporary devices. Whilst smartphones typically have more than enough memory for allocating graphics buffers, and additionally require such large memories for other purposes, such as video decoding and storing third-party application data, there are potentially other classes of portable device, which for reasons of cost or specialisation, might not reasonably include such large memories. If such a class of device were to exist, or emerge, then our approach would provide a mechanism for delivering high-resolution user interface rasterisation, without the need for megabytes of memory for video buffers, since each pixel can be generated on-demand and in rasterisation order, for direct consumption by the display hardware.

Interprocessor bandwidth

A phenomenon which emerges from our approach is the raising of the abstraction at which communication occurs between the CPU and the pipeline coprocessor, when compared to the contemporary approach for rendering on a GPU. Since the circuitry for applying synchronisation updates to the pipeline is itself synthesised, the syntax and semantics of the communication protocol are under complete control of the OPP generation processor. Consequently the number of values which are synchronised with the FPGA can be reduced. For instance the vertical offset of an entire view might be communicated, with the individual coordinates of each primitive vertex comprising the view derived within the FPGA. This stands in contrast to Android's current approach to GPU rendering of views, whereby the coordinates of each vertex within the user interface are streamed into the GPU.

If the magnitude of the interprocessor bandwidth were of concern then our approach has a clear advantage, however we have not found any evidence to suggest that power consumed by this form of communication is significant.

Coarse-grained reconfigurable architectures

Whilst FPGAs provide an accessible target for experimenting with place-and-route style optimisations, their power inefficiency is related to the granularity at which they can be reconfigured. Contemporary FPGAs tend to offer reconfigurability at the level of granularity of 1-6 bits: single bits, or small collections of bits, can be individually routed and operated upon. The price paid for a higher degree of flexibility is a reduced power efficiency. The kind of circuit which results from synthesising an OPP does not require such a flexible fabric as found in an FPGA. It seems quite likely that a fabric with granularity at the level of 8 or 16 bits would be the ideal target for our rendering application. Whilst this ideal target is not commercially available, there has been a growing interest in such coarse-grained reconfigurable architectures (CRGAs) in recent years, particularly within the domain of accelerators for deep neural networks [RMJ$^+$21].

This is particularly interesting, since the problem of place-and-route optimisation is largely orthogonal to the problem addressed in this chapter: obtaining the OPP configuration itself. If viable CGRAs emerge, perhaps for embedded acceleration of neural networks, we would expect the OPP we describe in this chapter to be implementable within such a CGRA, with considerably better energy efficiency than in an FPGA. In this sense the problem we are attempting to solve is of obtaining a representation of the rendering process (or related kinds of processes) in a form amenable to place-and-route optimisations, with the intention of targeting generalised spatial computing platforms.

4.10.2 Pipeline generation

The aim of our experiment was to generate an OPP from an application defined with respect to an API resembling Android as closely as possible. We did not achieve complete conformance with the Android API, for a number of reasons.

Firstly the languages and libraries which comprise the official Android distribution were either too complicated or too limited to adapt to the requirements of our experiment, within the amount of time we had available. Secondly the architects of Android made certain design decisions, motivated by particular historical factors, which made our experiment more difficult than necessary to conduct. Whilst we consider any divergences from Android to weaken the contributions of our experiment, we nonetheless believe they are justifiable—here we provide that justification, and where possible speculation on how future experiments might be brought into closer alignment with the official API.

Functional canvas API

The canvas API presented by Android is imperative in nature: the canvas object is mutable, and drawing operations are performed by side-effecting methods, which alter the canvas state. Whilst it is perhaps possible to generate a statically scheduled OPP for an application defined with respect to this imperative API, either using `translate` or some other mechanism, it presents certain challenges. Admitting operations which cause side-effects into the rasterisation process would require the translation system to keep a careful account of the temporal ordering of such operations, and to automatically infer data dependence relationships between operations based upon that ordering. Whilst this is a well studied problem in compiler design, in our opinion it is an *accidental complexity* introduced by spurious historical factors: as such extending our compiler to correctly handle this imperative API is not inherent to the experiment, and can be dealt with as a separable unit of research. We justify this assertion by observing how Android's use of its original canvas implementation and abstractions have changed over time.

As discussed in Section 4.3.1, originally Android executed most rasterisation operations on the CPU, rasterising primitives into a stateful bitmap buffer, immediately upon calling drawing methods of the respective canvas object. In the intervening years Android has switched to a primarily GPU based rasterisation model, whereby this original canvas implementation has been supplanted by a *recording canvas* implementation.

This recording canvas shares the same API, and is similarly stateful, but simply records the drawing operations performed on the canvas to its internal symbolic *display list*. As such the recording canvas is used solely to generate a symbolic representation of the user interface, serving as an intermediate representation which is then translated into a sequence of equivalent OpenGL commands, to be issued to the GPU. Note that if CPU-based rasterisation were still desired, then this could also be accomplished via the display list: the only operational difference observable being some additional temporary storage allocations for the display list, and a possible delay to the rasterisation of the applied primitives. This comes about because Android's canvas API is largely *write-only*—users of the canvas are not expected to inspect the state of individual pixels whilst the canvas is being mutated, and so immediate rasterisation is unnecessary.

Since rasterisation within Android has transitioned to using a symbolic canvas implementation, the supporting arguments in favour of an imperative API have become less relevant, whereby a functional interface would serve equally. With a functional canvas API, an *immutable* canvas object is supplied and returned from the draw method on view objects. These immutable canvas objects symbolically represent the appearance of the canvas, and admit efficient implementation since they are equivalent to a *linked list* of display list entries.

We defined the application of our experiment with respect to this kind of functional canvas API, since it provided practically equivalent functionality to the existing imperative API, appeared to be at least as suitable from a contemporary perspective, and simplified the implementation of our compiler.

Supporting imperative constructs

The compiler we developed for our experiment could only generate OPP fragments from reified methods which were both *pure* functions, and implemented exclusively using functional constructs internally. This is notably a stricter limitation than requiring the canvas API alone to be functional. In this section we consider the detrimental effect of this limitation, and how it might be overcome.

Since mutation, of at least *shared* state, is discouraged in Android within the rendering process, the requirement to implement methods responsible for drawing in a functional style is not in theory much of an imposition. It is however quite unnatural for Android developers to implement such methods using *only* functional constructs, especially in languages such as Java and Python, which have a relatively restrictive expression sub-language, compared to for instance a language like Haskell.

Although several imperative constructs are impossible to support in a statically scheduled OPP, for instance unbounded iteration, certain functional constructs are also unsupported, for instance unbounded recursion. Supporting translation of arbitrary source terms from any

Turing-complete language to a statically scheduled OPP is not theoretically possible—the issue is rather if the supported subset of the language is convenient or inconvenient.

Programmers used to an imperative language such as Python reasonably expect to use assignment statements. Whilst in general assignment statements can cause mutation of shared state, in many cases they are simply used for associating names with the result of certain intermediate calculations. When restricted to this latter application it is certainly possible to support the translation of such methods to an OPP. There are two strategies which we have considered: (i) syntactically transforming the original implementation into a functional form before translation, perhaps via conversion to static single assignment form (SSA), or (ii) extending our compiler internally to support constant propagation between statements. We briefly attempted approach (ii) with some encouraging results, but no serious attempt was completed. Subsequent to the experiment described in this chapter, we explored approach (i) in greater detail, developing a transformation named *let-weakening*—we describe let-weakening in Section 5.3.1.

Finally, in the particular case where it would be simpler to write a code generator for a method than rewrite the method functionally, or imperatively according to these restrictions, a code generator override for the method may be supplied explicitly.

View annotations

In Section 4.6.1 we introduced certain annotations which the application developer can apply to their view definitions, allowing them to constrain the behaviour of particular views more strictly than generally allowed by the class implementation. This was necessary for our experiment, since certain kinds of view in Android, such as recycler views, do not lend themselves to efficient rasterisation by an OPP in their full generality—for instance if a large number of children might potentially be added at runtime, the OPP must contain enough nodes to support this worst case, which can increase the latency and size of the pipeline enormously.

What is unclear is whether this burden on the application developer is either reasonable or necessary. Could an application developer reason about which constraints are necessary, without knowing the hardware capabilities of the application user's device, or without some feedback from the translation system? Could these constraints be inferred automatically, or be better supplied by the operating system?

Whilst we do not have a definitive answer to either of these questions, we imagine that at the very least some kind of reporting or debugging tools would be required to aid application developers in constraining their user interface effectively, since it is probably not obvious how much each degree of freedom costs in terms of pipeline resources. It seems likely that certain refinements to the standard views supplied by the operating system could reduce the burden placed on the application developer, but this is not something we attempted in our experiment.

Application profiling could potentially discover many useful constraints heuristically, such as the maximum observed population counts for the children of recycler views. Generating an OPP using speculative information would require confirming the constraint at runtime for each frame, with a fall-back mechanism to GPU- or CPU-based rasterisation should the constraint be violated. Again, this is an idea which was not explored in our experiment.

Text rasterisation

In Android, GPU-based rasterisation of text is accomplished by means of a font atlas. The Bézier curve rasterisation and anti-aliasing of each character is performed by the CPU, for each unique combination of glyph, style and size, and cached in the atlas, which is then synchronised periodically with a copy stored in the GPU texture memory, as and when new characters appear within the user interface. The GPU is simply responsible for compositing entries from the atlas onto the rasterised user interface, given a list of coordinates and their corresponding indices into the atlas. This architecture was probably selected since the complexity of implementing full font rasterisation in the GPU brings little benefit for typical user interfaces, since the atlas is only updated periodically, and serves as an effective cache, amortising the cost of CPU rasterisation over many frames. This stands in contrast to 3D rendering use cases, such as games, in which the projection of each glyph typically changes between every frame.

To implement text rasterisation in our FPGA based solution, an atlas-based architecture would perhaps serve also. However a shortcoming of our experiment was an incomplete implementation of text rasterisation—all text within our experimental user interface was rendered as a simple chevron pattern in place of the actual glyphs. Hence we can only speculate on the energy impact of text rasterisation on our experimental results, and how a more elaborate experiment might correctly handle text rendering.

A naïve implementation would introduce a separate node into the OPP for each text view in the user interface being rasterised. This approach is potentially quite inefficient, since each of these nodes would either require duplication of (some portion of) the atlas, or access to a shared memory with many concurrent read ports, in order to allow the pipeline to execute without stalling.

A more sophisticated implementation might separate out text composition from the OPP, using a parallel pipeline though which atlas indices stream through one single-ported copy of the atlas. The two resulting pixel streams might then be subsequently composited together as a post-processing stage. Since textual and non-textual elements potentially interact in complicated ways, the colour space of each pipeline might be extended to incorporate tagging bits, for conditioning the post-processor. A limitation of this approach would be an inability to handle overlapping glyphs, since it only supports a single set of indices being dereferenced into the atlas for composition with each pixel. Whilst many user interfaces are likely to be unaffected by this limitation, it might be mitigated either by implementing more read ports, copies of the atlas, or some CPU-based fall-back mechanism.

Implementation language

Throughout this chapter we have described our experiment with reference to an implementation in Python. We favour Python for OPP generation, over Java for example, since it readily supports the kinds of scope introspection we required. The officially supported languages of Android could in theory support the requisite reification operations, but would require further work. In the next chapter we describe in greater depth how such translations are generated from Python functions, generically and not in particular to OPP generation.

However the precise implementation of our experiment was rather more complicated than this. Most of the experiment was not carried out within Python, but instead within an interpreter for an imperative language of our own specification. This was done in part for pragmatic reasons, since prior to the experiment we did not know exactly what kinds of reified information would be required—it was easier to modify the implementation of a very simple interpreter in order to provide this information, as and where necessary.

This experimental interpreter was that of a minimal Lisp, although based upon F-expressions rather than S-expressions, as its fundamental form. The use of an F-expression based Lisp was motivated by their powerful support for metaprogramming and language extension. We relied heavily on the design of the Kernel programming language [Shu10], for constructing this interpreter. In retrospect, the full power of an F-expression based Lisp was unnecessary for generating the pipelines encountered in our experiment, and the same effects can be achieved in any programming language supporting reification of syntax and scope, such as Python. This we demonstrate in Chapter 5.

Consequently during our experiment we progressed from generating code within our custom interpreter to generating code within an equivalent system embedded in Python. Nevertheless, the majority of our results were established in the original experimental interpreter, and the Python version served only to prove that our techniques were independent of the peculiar nature of our experimental setup.

A couple of technical issues with our original compiler implementation were only identified whilst re-implementing it in Python. These were rectified in the Python implementation, however we did not back-port these fixes to the original system, nor re-evaluate all prior experimental data:

- Our original code generation system split translation of reified terms into two phases, firstly inlining constants, and subsequently performing code generation from these inlined forms. This phased strategy was found to be unnecessary, and removed from the translation system, resulting in the design described in this chapter.

- The initial mechanism we designed for the representation of non-immediate reified scopes was discovered to be ill-founded. This issue only manifested in a more general context, and not within that of our experimental application. In Section 5.6.4 we describe in depth the proper representation of non-immediate reified scopes, based upon *contextual generators*.

4.11 Related work

Whilst the relevant related work for the metaprogramming aspects of this chapter have already been surveyed in Chapter 3, it remains to discuss the related work pertaining to object-oriented rendering, and also offloading computation at runtime to reconfigurable coprocessors.

4.11.1 Object-oriented rendering

Gharachorloo et al. [GGSS89] characterise three distinct categories of rasterisation techniques, specifically frame-buffer, virtual-buffer and object-oriented based techniques. Their 1989 survey predates contemporary GPU architectures, however the broad characterisations they present are still applicable. Their treatment of frame-buffer and virtual-buffer rasterisation techniques focuses on optimisations for conventional random-access memory systems, based upon memory access patterns, and also custom memory architectures, which in some cases bestow the memories with specialised computational facilities. What they term *object-oriented* techniques best aligns with the approach to rasterisation taken in this chapter. Within object-oriented techniques they identify two broad subcategories: (i) sprite-based techniques; and (ii) processor-per-polygon based techniques. What these two radically different approaches share in common is only the *manifestation of graphical abstractions* within the physical rasterisation architecture. They characterise processor-per-polygon rasterisation specifically as assigning each polygon within the rasterised image to a separate physical processor. These processors are chained together to form a linear pixel-wide pipeline. Their work implicitly assumes, as it predates the availability of high-capacity FPGAs, a fixed-function pipeline implementation, with variability only in the assignable parameters of the polygon processors. They only consider processors capable of rendering polygons, and not of any other kind of generic object, or application-specific object. Foley et al. [FvDFH96] present a similar survey, although they do generalise their characterisation, using the generic term of processor-per-primitive pipelines.

Various interesting, concrete instances of object-oriented rasterisation architectures have been explored in the literature, notably: (i) Locanthi [Loc79] describes a specialised MOS/LSI integrated circuit, taking into account the gate-level silicon layout, for rasterising a collection of rectangles; (ii) Schneider and Claussen [SC88] define a complex pipelined architecture, from which we take the term object-processor pipeline (OPP), which includes post-processing stages for shading and filtering the image; (iii) Weinberg [Wei81] describes a pipeline specifically suited to generating antialiased images; (iv) Fussell and Rathi [FR82] present a real-time shaded triangle-based processor system, paying specific attention to the performance attained for a fixed-cost when implemented using contemporary VLSI technology; and (v) Deering et al. [DWS+88] define a sophisticated triangle-based pipeline architecture specialised for 3D rendering, with fixed-function shading hardware supporting multiple light sources.

4.11.2 Dynamic offloading to reconfigurable coprocessors

A wide array of diverse techniques and systems for offloading computations to reconfigurable coprocessors have been proposed. Notable approaches, which focus on dynamic compilation or customisation of the coprocessor include: (i) Lysecky et al. [LSV04] describe a processing architecture which employs an FPGA as a coprocessor, to improve the performance and reduce the energy consumption of software running on a microprocessor—the executing program is profiled at runtime, and critical regions are JIT compiled and linked to an FPGA configuration; (ii) Ma et al. [MAA16] present a technique for dynamically assembling FPGA accelerators from pre-synthesised and partially reconfigurable tiles, dynamically modifying the FPGA LUTs, which are arranged in a 2D array—their approach boasts orders of magnitude improvement in compilation times compared to full runtime synthesis, and is not based on profiling, but instead includes a DSL which implements certain primitives within the accelerator; (iii) Capalija and Abdelrahman [CA11] define a similar approach, but based on dynamic profiling to identify commonly executed code fragments, which reconfigures the state of a pre-synthesised FPGA overlay; (iv) Clark et al. [CBC+05] describe how the instruction set of a microprocessor can be dynamically extended to include custom instructions which implement dataflow subgraphs for critical sections of application code—these custom instructions are synthesised and linked at runtime, and implemented in a configurable compute accelerator (CCA) which comprises an array of functional units connected in a feed-forward network; and finally (v) Bergeron et al. [BFD08] present an approach similar to Clark, which instead targets a commodity FPGA.

4.11.3 Low-power GPU computing

The extent to which mobile GPUs already offer a power-efficient computing platform, compared to mobile CPUs, has been the subject of a number of studies: (i) Mittal and Vetter [MV14] provide a broad and detailed survey of the state of the art, with attention to GPU energy-efficiency in particular; (ii) Maghazeh et al. [MBEP13] implement a number of benchmarks to assess the power efficiency of computing on mobile GPUs—they find that GPUs show promise as an energy-efficient computing platform, but necessitate different optimisation strategies to their non-mobile counterparts; and (iii) Grasso et al. [GRR+14] assess the suitability of the ARM Mali GPU embedded in an Exynos 5250 SoC, specifically for 64-bit floating-point high-performance computing (HPC) workloads, finding it simultaneously faster and lower-power than the corresponding ARM CPU within their HPC benchmarks.

4.12 Conclusion

In this chapter we have applied the technique of late-bound code generation to the problem of offloading the rasterisation of a smartphone user interface to an FPGA coprocessor. The solution we presented consists of two independent parts: (i) an application-specific object-processor pipeline rasterisation architecture, and (ii) a code-generation system capable of translating a source program written in terms of contemporary user-interface abstractions to an instance of this architecture. Our rendering architecture is a deeply pipelined dataflow graph, instances of which are heavily specialised to the given application and its user interface. Our code generation system is capable of accommodating user interfaces which feature a combination of both standard and programmatically defined components.

Although the pipeline generated for a particular user interface is highly application-specific, it still accommodates the full range of variability which the user interface might span. This variability is materialised within the pipeline as a number of mutable registers, which may be read and written by the host processor, in order to synchronise the state of the pipeline with the state of the application. Incorporating this variability into the pipeline is crucial, since it is impractical to generate and independently specialise a pipeline for every frame. Our state synchronisation process is efficient, since these registers need only to be updated once, in batch, between frames.

Our pipeline architecture corresponds to a fully inlined and fully unrolled implementation of raster drawing procedure, and is furthermore completely statically scheduled and amenable to place-and-route optimisation, motivated by the objective of minimising the power consumed due to data communication. In principle it appears to be capable of handling the structure of user interfaces substantially more complex than that of our experimental application, although it is inherently designed for 2D graphics, and is not suitable for 3D graphics, at least in its current form. This deeply pipelined design makes effective use of the FPGA, since for the majority of active clock cycles the entire circuit is usefully switching, and therefore not wasting energy solely due to static leakage currents.

The architectural decision to admit only a statically scheduled pipeline was motivated by the objective of eliminating dynamic control circuitry, which contributes to the power consumed by a circuit, and if eliminated could give rise to a power saving. Whilst we succeeded in generating a statically scheduled pipeline for our experimental application, it was necessary to introduce a significant number of deep queues into the resulting circuit, required in order to maintain data synchronisation. It is unclear whether the power consumed by these queues is potentially greater than that of the control circuitry of a dynamically scheduled pipeline, designed to trade a reduction in queue depth for a modicum of dynamic control.

In order to generate practical pipelines from contemporary user interface abstractions, such as those present in the Android operating system, it was found necessary for the application developer to annotate their user interface description with additional metadata. These annotations typically place static limits on the range of behaviour the user interface might exhibit, allowing the pipeline generator to implement significantly simpler and more specialised pipelines. Although these annotations appear to be of an acceptable kind, in that they are clearly understandable to an application developer, it would be necessary to provide additional tooling to guide developers as to which elements of their user interface are in need of further annotation—especially in those situations where a feasible pipeline could not be generated at all. Alternatively an automatic annotation system, perhaps even speculative in nature, could be envisioned.

Although our pipeline generation system was capable of handling user interface abstractions typical of contemporary smartphones, further work would be required in order to handle the imperative drawing interface currently presented by the Android operating system. Our experimental application was defined with respect to a similar, but functional interface. Furthermore, our system places a considerable burden on the smartphone operating system developers, requiring considerable work in implementing RTL components for the rasterisation of each primitive graphical operation—and perhaps even extending to hand-optimised RTL implementations of each standard user interface component. Significant additional work would also be required to integrate the system we have described into its eventual hardware and software context—perhaps even necessitating standardisation of the interface presented by the FPGA—and finally to develop the text rasterisation compositor we outlined, but did not implement.

With respect to power consumption, we were unable to effectively compare the power consumed by our pipeline with that consumed within the GPU coprocessor of a contemporary smartphone, attributable to the rasterisation of a similar user interface. This makes it very difficult to put our power consumption results in context. These power consumption results are themselves estimates, and have a high degree of error: for instance they neglect various factors, including the energy consumed in performing the place-and-route optimisation, and in programming the FPGA. As an absolute power consumption value, without any comparison to a similar application executing on a contemporary smartphone, the value itself is not particularly suggestive, since our experimental application is extremely simple compared to a typical smartphone application. In short, the power consumption of our pipeline is not sufficiently low to be compelling in and of itself.

Chapter 5

ReiPy: late-bound code generation in Python

5.1 Introduction

In this chapter we describe the design of *ReiPy*, a library for performing late-bound code generation within the Python programming language. ReiPy consists of four components: a preprocessor; a front-end reifier; a meta-circular interpreter; and a modular set of code generators, which translate terms accepted by the interpreter into terms of the target language. In Sections 5.3 though 5.6 we describe the design and implementation of each component of ReiPy in turn, without reference to any particular application. This is done in order to provide a concrete basis for the technical contributions of this thesis. Finally, in Section 5.7 we illustrate the system as a whole, applying it to the problem of offloading fragments of computation to the "cloud".

To simplify our description, throughout this chapter we consider only a singular target language: Python itself. Furthermore, the system as described is not currently capable of translating arbitrary Python definitions, spanning the full programming language syntax or semantics. Some notable omissions include lack of support for: early function return; throwing or catching exceptions; coroutines or generators; variable shadowing; or non-positional arguments. In

Section 5.8 we discuss these limitations, and how they might be overcome with further work.

5.2 Motivation

Translating the behaviour of Python functions to Python terms is not a redundant operation, as it might so appear at first glance. Consider for example the problem of determining which values stored in a key-value database hash to a particular value. If the query language provided by the database does not support native calculation of the requisite hash function, computation of the hash would need to be performed externally to the database.

The following Python function might be used to extract a value from the database, compute the hash of the value, and then compare it against the hash of interest:

```
def predicate(some_key, some_hash):
  value = DB.query('GET KEY ' + some_key)
  hash_result = HASH(value)
  return hash_result == some_hash
```

Here we assume that `DB` is bound to a database connection object, and `HASH` is bound to the relevant hash function. Invoking `predicate` with a particular key and hash value will return `True` or `False`, depending on whether the value mapped to that database key hashes to the given value, or not.

Suppose that `predicate` has been defined in a Python interpreter executing on a different machine to the database server. In this case a *complete* representation of `value` must be communicated between the server and the interpreter, perhaps over a network. If the size of the value is considerable, for instance several gigabytes, it would be much more efficient for the Python function to be invoked in an interpreter executing on the database server itself.

Simply sending the literal syntax of the Python function definition to the remote server, with its argument values, and then evaluating it there, is not sufficient to offload the computation.

This is because `predicate` is a closure, and the concrete syntax of its definition lacks information about how `DB` and `HASH` are bound. To offload the computation as a Python term, whilst preserving the behaviour implicitly due to closure bindings, we are left with two options: (i) establish an effectively identical scope in the target interpreter, in which to evaluate the original concrete syntax of the function definition; (ii) create an equivalent Python fragment to the original definition, but restricted to some base set of terms already agreed upon by both interpreters. Below we illustrate a solution of the second kind, making use of ReiPy's translation capabilities.

Here we apply `translate` to the predicate function, providing a symbolic key argument, namely `argument_0`, and an immediate value for the hash argument:

```
interesting_hash = 0x276104A8
translation = translate(PythonBlock,
  immediate(PythonBlock, predicate, mandate), [
    PythonBlock([], PythonLoadName('argument_0')),
    immediate(PythonBlock, interesting_hash, mandate)
  ], mandate
)
```

This yields the following translation:

```
V0 = LocalDatabaseConnection('MODE=uds;PATH=/tmp/database.sock')
V1 = V0.query
V2 = 'GET KEY '
V3 = V2 + argument_0
V4 = V1(V3)
V5 = zlib.adler32
V6 = V5(V4)
V7 = 660669608
return V6 == V7
```

Note that the terms `DB` and `HASH` do not appear in the resulting term. The only names within the translation which close over its evaluation scope are: `argument_0`; `zlib`; and `LocalDatabaseConnection`. The latter two are bound to Python's standard library `zlib` module, and our database's connection class respectively. The user is required to ensure that these terms are agreed upon by both the original and target Python interpreter, perhaps due to a common import header:

```
import zlib
from MyDatabaseDriver import LocalDatabaseConnection
```

If these terms were not in common, then ReiPy could be used to represent the function's behaviour according to some different set of terms. For instance, if only the basic terms of the Python language were agreed upon, then ReiPy could in principle be employed to emit the implementation of `zlib.adler32` and `LocalDatabaseConnection` with respect to only these basic terms.

In this example, whilst the name `HASH` in our function definition was bound to the builtin Python hash function `zlib.adler32`, `DB` was *not* bound to an instance of the class `LocalDatabaseConnection`. To illustrate ReiPy's generality, we configured our `mandate` object to substitute the original TCP/IP connection object with an equivalent, but more efficient, Unix domain socket connection instead:

```
mandate = MyMandate()
mandate.map_remote_to_local_database(
  'MODE=tcp;HOST=database.example.com', 'MODE=uds;PATH=/tmp/database.sock'
)
```

We will return to this example again in Section 5.7, and explore in depth how this translation was accomplished. In the following section we begin our description of ReiPy's design and implementation in abstract terms, without reference to any particular application, starting with its front-end preprocessor.

5.3 Preprocessor

Given the modular nature of our code generation system—which we outlined in Chapter 2—we must be careful in our treatment of source terms which assign to potentially mutable variables. In imperative languages such as Python, information is typically retained and transmitted between operations indirectly via named, shared, mutable and lexically scoped variables. Whilst these variables are often local to a function invocation, other times they might be global, bound to a class or object instance, or perhaps some other kind of block-level scope. This mechanism of exchanging and persisting information throughout the execution of a program is simple and elegant, yet from the perspective of a compiler it is frequently found rather more powerful than strictly required, at least within a specific context.

For example, whilst variables might in general be (i) captured within the closure scope of a first-class function, (ii) be assigned to from several distinct assignment statements, or (iii) be reassigned again and again over time, we note that particular variables within a given program rarely take advantage of all of these features simultaneously. An optimising compiler must identify, on a case-by-case basis, which features a particular variable makes use of, in order to generate code which elides the overhead incurred by the loose abstraction that general purpose variables present.

Put another way, writing an algorithm in terms of stateful variables, but which does not exercise its right to mutate those variables, requires a compiler to prove the impossibility of variable mutation in order to generate a translation absent of mutable state. A similar situation arises with the allocation and deallocation of variables which might be captured by a closure, et cetera.

Although it might be infeasible for a compiler to accurately prove *exactly* which subset of features are required for every variable, it is certainly feasible for a compiler to identify certain common usage patterns, for instance to reduce the extent of mutable state within the target program. It is often possible to identify certain patterns of variable usage which never lead to reassignment, or ever appear in the scope of a closure.

In this thesis we are particularly interested in generating pipelines for accelerating particular functions, specifically from Python definitions. To support the general behaviour of Python variables, our pipelines would require heap allocated, garbage collected mutable state for each variable instance created or accessed by the accelerated function. For certain interesting execution targets such as FPGAs, this would likely obliterate any chance of substantially accelerating the function, since the pipeline would incur frequent accesses to shared memory. Furthermore, such interactions also reduce the feasibility of finding a static schedule for the pipeline, if one is so desired. For this reason it is particularly important for ReiPy to identify certain common usage patterns of variables, in order to reduce or eliminate the presence of shared mutable state within its target pipeline.

In the following section we describe our solution for propagating information related to named variables between modular code generators, whilst simultaneously eliminating unnecessary mutable state from the target program. This optimisation is optional, although translations which are generated by ReiPy without it are typically unsatisfactory.

5.3.1 Let-weakening

Appel has noted [App98], based on prior work by Kelsey [Kel95], that the structure of an imperative program in SSA form [AWZ88, RWZ88] suggests an equivalent *functional* program. Functional in the sense that all mutable variable bindings in the original program are replaced by immutable bindings within the corresponding program. The functional program eliminates variable mutation using a nest of lexically scoped closures, whereby rebinding is instead achieved by recursive function invocation. Whilst this transformation is incredibly powerful, fully accommodating both non-structured control-flow and SSA ϕ-nodes, we observe than certain kinds of program can benefit from a simpler, specialised form of this transformation.

Assuming structured control flow, if all the uses of a particular variable definition are reached *only* by that definition, and those uses occur only within statements following the assignment (subsequently within the same block, or lexically nested, excluding closures) then the mutating assignment may be transformed into an immutable let-binding, without any need to introduce a recursive continuation. All other instances of mutating assignment statements, which may reassign to a variable, are left untransformed. We term this specialised transformation *let-weakening*.

Put another way, an assignment is disqualified from let-weakening if either: (i) another intervening assignment to the variable *could occur* (in time) between the candidate definition and its uses; or (ii) some uses of the definition are outside the scope of the candidate let-binding.

Structurally let-weakening is a transformation which replaces sequences of statements of the form:

```
[statement_i-1]
[variable_name] := [expression]
[statement_i+1]
[statement_i+2]
...
[statement_i+n]
[statement_i+n+1]
```

with the form:

```
[statement_i-1]
let [unique_name] = [expression] in:
  [statement_i+1]
  [statement_i+2]
  ...
  [statement_i+n]
[statement_i+n+1]
```

The mutating assignment (`:=`) to `variable_name` is replaced with a let-binding to `unique_name`, which immutably binds `unique_name` to the result of evaluating the same expression, within and only within an inner block scope. The string `unique_name` is chosen uniquely to prevent any possible conflict with other names. All related uses of `variable_name` are replaced by uses of `unique_name` in `statement_i+1` to `i+n`.

An assignment which has been let-weakened offers two benefits to the subsequent code generation process. Firstly the assignment operation may be translated without involving any *mutation* to shared state, since all unshadowed uses of the name within the block necessarily refer directly to the evaluation of `expression`. Secondly all unshadowed uses of the name within the block can then be translated without incurring any *accesses* to shared state. For example, if the translation target is an FPGA pipeline, this means that corresponding definitions and uses of let-weakened assignments may be implemented as FIFOs, simply forwarding values from the evaluation of each `expression` to its use(s), without any dynamic accesses to shared memory. ReiPy's current design does not support variable shadowing within the source program, further simplifying its implementation.

Analysis

Whilst an analysis for let-weakening is likely possible using only reaching definitions, we follow Appel and base our analysis on a typical SSA conversion. The danger of let-weakening a name ineligible for let-weakening, is that the name is bound too early: uses of the variable might end up resolving to an earlier assignment than in the original program. As such any assignments for which the definition can outlive the termination of the let-in block are immediately and conservatively disqualified from let-weakening.

For this reason, ReiPy begins its let-weakening analysis by identifying all names assigned within a function definition which correspond to *stack-allocated local variables*—or equivalently, all variables local to the function invocation which are not captured in a closure. Fortunately we may simply reuse Python's own determination for this set, which we shall see in Section 5.4.1 are accessible via the `co_varnames` attribute, on any compiled bytecode object. Helpfully, Python provides a "builtin" function `compile` for generating bytecode from concrete syntactical definitions, if required.

A control-flow graph (CFG) for the function definition is built by recursively walking its abstract syntax tree, adding “guard” nodes before any primitive operations which could raise an exception. These guard nodes have outward edges to every outer exception handler, and also the terminal node of the graph. No attempt is made at interprocedural analysis, and so invocations are represented simply as nodes within the CFG. The CFG is stripped of unreachable nodes, which would otherwise disrupt our determination of dominance relationships.

Using a worklist, initialised to contain only the start node of the CFG, the nodes dominated by a given node are determined as those nodes which are unreachable if the given node is removed from the graph. Inverting this relation yields the set of dominators for each node. The immediate dominator for each node is found using the following naïve Python algorithm, where `nodes` is the set of all nodes in the CFG, and `dominators[node]` is the set of dominators for `node`:

```python
immediate_dominator = {}
for subject in nodes:
  candidates = set()
  for node in dominators[subject] - {subject}:
    rejected = False
    for other in dominators[subject] - {subject, node}:
      if node in dominators[other] - {other}:
        rejected = True
    if not rejected:
      candidates.add(node)
  assert subject is initial_node or len(candidates) == 1
  if subject is not initial_node:
    immediate_dominator[subject] = candidates.pop()
```

The dominance frontier for each node is then determined using an algorithm due to Cooper et al. [CHK01]:

```
dominance_frontier = {}
immediate_predecessors = {}
for node in nodes:
  dominance_frontier[node] = set()
  immediate_predecessors[node] = set()
for node in nodes:
  for successor in node.successors:
    immediate_predecessors[successor].add(node)
for node in nodes:
  predecessors = immediate_predecessors[node]
  if len(predecessors) > 1:
    for predecessor in predecessors:
      cursor = predecessor
      while cursor is not immediate_dominator[node]:
        dominance_frontier[cursor].add(node)
        cursor = immediate_dominator[cursor]
```

A classic live variable analysis (LVA) is performed, finding the fixed-point of the following pair of dataflow equations, in order to determine the presence of ϕ-nodes within a *pruned* SSA form:

$$\mathrm{LIVE}_{\mathrm{out}}[x] = \bigcup_{y \in \mathrm{succ}(x)} \mathrm{LIVE}_{\mathrm{in}}[y] \tag{5.1}$$

$$\mathrm{LIVE}_{\mathrm{in}}[x] = \mathrm{USED}[x] \cup (\mathrm{LIVE}_{\mathrm{out}}[x] - \mathrm{ASSIGNED}[x]) \tag{5.2}$$

$\mathrm{LIVE}_{\mathrm{out}}[x]$ and $\mathrm{LIVE}_{\mathrm{in}}[x]$ are sets containing all variable names potentially live at the exit and entry points of node x respectively. $\mathrm{USED}[x]$ is the set of all variable names used within node x before being assigned, whereas $\mathrm{ASSIGNED}[x]$ is the set of all variable names assigned within node x. The function $\mathrm{succ}(x)$ simply returns the set of all successor nodes to x within the control-flow graph.

We populate our initial set of candidates for let-weakening with all assignments to stack-locals which are dead (i.e. not live according to LVA) in their dominance frontier, and for which all other assignments to the same name dominated by the candidate are also dead in their dominance frontier. This is simply the set of all assignments, in pruned SSA form, which are not used within any ϕ-node. For each candidate in this set we find all of its uses: specifically all uses dominated by the assignment which are not dominated by any other assignment dominated by the candidate.

Finally we filter out any candidates which have uses outside the scope of the block let-weakening would introduce. By default the scope of the block is extended maximally, to encompass all the following statements within the block of the assignment. If an assignment has uses in lexically enclosing blocks, this filtering step disqualifies them from let-weakening.

5.4 Front-end reifier

Late-bound code generation is, by definition, performed within the runtime context of the source program. Consequently, any system for late-bound code generation must come equipped with some form of *reifier*: a mechanism to materialise terms which comprise the source program's implementation, into values which are addressable within that source program's runtime context.

Python does not have a well-established formal semantics, and so it is ambiguous whether the implementation of our reifier relies solely upon the semantics of Python, or in part upon various implementation details of the CPython 3 interpreter.

In general, in order to correctly translate the behaviour of a runtime function, three pieces of information are required: the syntax of the function definition; the scope over which the function instance is closed; and the semantics of the language in which the definition is denoted. Intuitively, the closure scope gives meaning to all free variables within the function's syntactical definition, and the language semantics gives meaning to the remainder of the syntax, such as keywords and punctuation. ReiPy obtains the first two of these pieces of information by means of reification.

Whilst syntax reification is perhaps a more common operation than scope reification, it is curious that in Python reification of scope is vastly more straightforward than reification of syntax.

5.4.1 Scope

Python scopes are formed of four parts, in order of decreasing priority: the local scope, the closure scope, the global scope, and ultimately the “builtin” namespace. When resolving a variable by name, first the local scope is checked, and if a variable with that name exists in the local scope, the name resolves to the value of that variable. Failing that, the closure scope is checked, which might be nested in general, and perhaps absent, then the global scope, and finally Python’s list of “builtins”. If no variable with that name is found in any of the four levels of scope, a `NameError` is raised. A consequence of this arrangement is that loops do not establish a scope separate from that of the enclosing function, since there is no higher priority scope than the local function scope—statements executing in a loop body execute within the same scope as statements of the same function residing outside the loop body. One peculiar complication arises from Python’s use of *variable hoisting*, whereby it implicitly creates a binding in the local function scope for any name which is *possibly assigned* within that function. This behaviour may be modified on a case-by-case basis, by declaring the variable global or non-local using the `global` and `nonlocal` keywords. Notionally this binding is created before the very first statement of the function’s body is executed, hence the term *hoisting*, with the variable left uninitialised: any attempt to read the variable raises an `UnboundLocalError`. Modules and classes both introduce scopes into the scope hierarchy, however these manifest either at the level of global scope, local scope, or within the nest of closure scopes.

In order to reify the lexical scope of a Python function instance, just three pieces of information are required, all of which are available as attributes of the function object. Python maintains a fairly comprehensive selection of information about the scope of any function, as the following interactive shell excerpt illustrates.

```
Python 3.7.3 (default, Jul 25 2020, 13:03:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> i = 123
>>> def outer(a):
...     j = 456
...     x = 'abc'
...     def subject(b):
...         y = 'xyz'
...         z = 'foo'
...         def inner(s):
...             return z + s
...         print(a + b + i + j)
...         print(inner('bar') + x + y)
...     return subject
...
>>> f = outer(789)
>>> f(0)
1368
foobarabcxyz
>>> f.__code__
<code object subject at 0x7f0e4aa2f660, file "<stdin>", line 4>
>>> f.__code__.co_cellvars
('z',)
>>> f.__code__.co_freevars
('a', 'j', 'x')
>>> f.__code__.co_varnames
('b', 'y', 'inner')
>>> f.__closure__
```

```
( <cell at 0x7f0e4aa18a08: int object at 0x7f0e4a9f2830>,
  <cell at 0x7f0e4a9e2618: int object at 0x7f0e4a9f27b0>,
  <cell at 0x7f0e4a9e26a8: str object at 0x7f0e4aab5538> )
>>> f.__closure__[0].cell_contents
789
>>> f.__closure__[1].cell_contents
456
>>> f.__closure__[2].cell_contents
'abc'
>>> f.__globals__
{ '__name__': '__main__',
  '__doc__': None,
  '__package__': None,
  '__loader__': <class  '_frozen_importlib.BuiltinImporter'>,
  '__spec__': None,
  '__annotations__': {},
  '__builtins__': <module 'builtins' (built-in)>,
  'i': 123,
  'outer': <function outer at 0x7f0e4a9987b8>,
  'f': <function outer.<locals>.subject at 0x7f0e4a998840> }
```

As demonstrated by the above excerpt, code attributes such as `__code__.co_varnames` and `__code__.co_cellvars` pertain to bindings to be instantiated within the local scope of the function, and the scope of any functions nested within, such as `inner`. Notice also that the builtins namespace is linked using named indirection through the global scope, via the variable named '`__builtins__`'.

Since at least Python version 3.7, it is possible to both *read* and *write* the `cell_contents` attribute of the cell object, to resolve and assign to the corresponding variable, respectively: enabling a fully *reflective* materialisation of function scope. In earlier versions of Python it was necessary to call `PyCell_Set` via the `ctypes.pythonapi` foreign-function interface, mutating the internal state of the Python interpreter, in order to reflect changes made to the reified scope back into the runtime. The standard Python library `inspect` provides a somewhat more abstract interface to scope bindings than does the `__code__` attribute, but to our knowledge it does not support the mutation of bindings—hence it cannot be used if a reflective reification is desired, as it is in ReiPy.

Since we are only interested in reifying the scope over which the function is closed—namely the lexical scope in which the function was defined—many of these attributes are superfluous. We only require three attributes: `__code__.co_freevars`, `__closure__` and `__globals__`. The first two of these attributes are tuples of exactly the same length, since the former contains the names associated with the cells in the latter (although in the case that `co_freevars` is an empty tuple `__closure__` is `None`). This association can be observed in our example above, in which `f.__code__.co_freevars[2] == 'x'`. From the function definition we expect that `x == 'abc'`, and indeed we observe `f.__closure__[2].cell_contents == 'abc'`.

Accordingly ReiPy reifies the lexical scope of any Python function thus:

```
def reify_scope(function):
  if not hasattr(function, '__cached_closure_scope__'):
    assert callable(function) and hasattr(function, '__code__')
    function.__cached_closure_scope__ = PythonClosureScope(
      function.__code__.co_freevars,
      function.__closure__ or (),
      function.__globals__
    )
  return function.__cached_closure_scope__
```

We define the constructor for `PythonClosureScope` as follows:

```
class PythonClosureScope(PythonScope):
  def __init__(self, names, cells, global_namespace):
    names = names if names is not None else ()
    cells = cells if cells is not None else ()
    assert isinstance(names, tuple)
    assert isinstance(cells, tuple)
    assert isinstance(global_namespace, dict)
    assert len(names) == len(cells)
    self.__bindings = dict(zip(names, cells))
    self.__global_namespace = global_namespace

  ...
```

The parent class `PythonScope` is simply an abstract class shared amongst other forms of reified scope, which we shall encounter later. The minimal interface for objects of class `PythonScope` is:

```
resolve(self: PythonScope, name: str) -> object
```

The method `resolve` simply returns the object bound to `name` in the given scope.

As we have seen, closure scopes in Python defer resolution of unbound names to the dictionary of global variables, and failing that to the builtin namespace. Accordingly the method `resolve` is defined on `PythonClosureScope` as follows:

```
def resolve(self, name):
  if name in self.__bindings:
    return self.__bindings[name].cell_contents
  else:
    try:
      return self.__global_namespace[name]
    except KeyError:
```

```
from types import ModuleType
builtins = self.__global_namespace['__builtins__']
if isinstance(builtins, ModuleType):
  try:
    return builtins.__dict__[name]
  except KeyError:
    raise Exception('name "' + name + '" is unbound')
else:
  try:
    return builtins[name]
  except KeyError:
    raise Exception('name "' + name + '" is unbound')
```

In CPython, the object bound to `__builtins__` within the global dictionary may be either a module, or a plain dictionary, depending for example on whether the function was defined outside of the program's main module. Accordingly both cases must be handled, since when the builtins namespace is implemented using a module, the dictionary of bindings is accessed via its `__dict__` attribute.

`PythonClosureScope` conforms to the interface required for scopes by the meta-circular interpreter defined in Section 5.5.

5.4.2 Syntax

Perhaps surprisingly, reification of the syntax of Python function definitions is technically more challenging than the reification of their scope. The CPython interpreter does not persist the concrete or abstract syntax fragment from which a function was instantiated. If the function was defined from syntax originally read from a file, the `__code__.co_filename` and `__code__.co_firstlineno` attributes of the function potentially record the path at which that file was located on the filesystem at the time the function was allocated, together with the first line within which the definition begins.

Prior to Python version 3.9, `co_filename` was a path relative to the current working directory at the time the function was instantiated, and the absolute path of the current working directory was not recorded, so it was not possible in general to resolve the file—since the current working directory might have changed in the meantime. Since version 3.9 however the path recorded by `co_filename` is always an absolute path [Fou19]. Nevertheless, the contents of the file at that path might have been modified since the function was instantiated, and therefore any function definition found within that file is not necessarily related to the definition of the function instance. Furthermore, due to optimisation details internal to the CPython interpreter the `co_filename` property is not populated reliably, however the path may be somewhat more reliably obtained by using the `inspect.getsourcefile` library function, which performs more extensive, yet still heuristic, inferences.

ReiPy simply assumes that the contents of the file so obtained has not changed since the function was instantiated, and thus might produce unsound results if that assumption is violated.

Having assumed the contents of the file containing the function definition, together with the line number of the first line of the definition, typically provides sufficient information to identify the respective concrete syntax fragment. However ambiguity can arise if several function definitions begin at the same line. Fortunately Python's grammar prevents several *named* function definitions from beginning at the same line, but no such constraint holds for anonymous functions instantiated using the `lambda` operator.

Consequently ReiPy attempts to reify the syntax of the function definition in general using a disambiguation strategy, due to Ristin [Ris18]. In order to reify the syntax of the function definition we parse the concrete syntax of the entire file, using the standard `ast` library, rendering a module-level abstract syntax tree. Walking the tree, nodes corresponding to each function definition are obtained, and then filtered by line number. In the case that there is only one definition on that line, then the singular node serves to reify the function's syntax. In the case of ambiguity each candidate definition is converted to CPython bytecode using the `compile` builtin, and compared for equivalence against the `__code__.co_code` attribute of the function being reified. Identical bytecode sequences are trivially equivalent, however the CPython compiler might compile the same concrete definition into a range of different equivalent sequences. Establishing the equivalence of two programs is undecidable in general, so we proceed heuristically. The bytecode for each function definition on that line is compared to `__code__.co_code` modulo a small number of synonymous rewrites. If only one definition matches, then the AST of that definition is used, otherwise an exception is raised.

Assuming that a unique native AST for the function is obtained, we build a `PythonTerm` object, ReiPy's own representation of syntax—this is for a number of technical reasons, the most important of which is that the design of our meta-circular interpreter requires terms to conform to this `PythonTerm` interface, and not Python's native `ast` module, as we shall see later.

ReiPy can translate terms from Python to Python itself, which is useful for reasons of optimisation and distribution as we illustrate in Section 5.7, and in doing so uses `PythonTerm` for representing both source and target syntax. As we discussed in Chapter 2, our translation system requires the target (intermediate) representation to accommodate references to instances within the source program runtime, which `PythonTerm` does, yet Python's native `ast` module does not. Another reason for our own representation, is that `PythonTerm` only permits terms absent of early function return, whereas Python's native `ast` module allows definitions containing early return statements. This current limitation greatly simplifies the translation system.

The minimal interface for `PythonTerm` objects is as follows:

```
evaluate(self: PythonTerm, scope: PythonScope) -> object
is_immediate(self: PythonTerm) -> bool
immediate_reference(self: PythonTerm) -> object
```

The `evaluate` method simply evaluates the term in the given ReiPy scope, possibly returning an object. Terms for Python *expressions* evaluate to the value of the expression, whereas terms for Python *statements* typically evaluate to `None`, except blocks which evaluate to the value of the returned expression, if the last statement is a `return` statement. We describe the behaviour of the methods `is_immediate` and `immediate_reference` later in Section 5.5.

Currently ReiPy does not supply the requisite rewrite transformation for converting function definitions containing early returns to equivalent definitions without. Whilst this could be defined, it might instead be simpler to properly support early returns within the code generation procedure. Until either is made available, ReiPy is unable to handle the Python `return` keyword in any position other than the last statement of the function body. Similarly ReiPy does not support Python's `try` or `raise` keywords.

The AST of the definition is walked recursively, creating an instance of `PythonTerm` for each node in the tree. The process is a transliteration, however it is at this stage that let-weakening is performed. After having performed the let-weakening analysis on the native AST, as was described in Section 5.3.1, those let-weakening annotations are checked during instantiation of the `PythonTerm` based representation: whenever an assignment is encountered which is eligible for let-weakening, a special `PythonLetIn` term is instantiated, rather than a regular assignment term. Since Python does not support any form of let-in construct, the language represented by `PythonTerm` must be seen as an extension of regular Python. Since early returns and exceptions are not supported by `PythonTerm`, it is simultaneously a specialisation of regular Python. Let-in bindings are not simply syntactic, but introduce their own kind of scope, represented in ReiPy as an instance of `PythonLetInScope` which extends the base `PythonScope` class. `PythonLetInScope` is rather trivially implemented, as it forwards all operations on names to the parent scope, expect for resolutions to the let-in bound name, which is resolved to the bound value, and non-reassignable.

When instantiating a `PythonLetIn` term, all subsequent statements within the current block are moved into the let-in body. The name of the binding is generated uniquely, appending an integer to the original name such that it doesn't interfere with any other name in scope. All uses of the let-in binding are renamed to match the generated name.

This procedure for syntax reification is implemented by ReiPy's `reify_syntax` function:

Algorithm 1 Reify syntax

```
Require: f is a function
Require: f is not a method
  Parse the file containing the definition of f using the ast library;
  Find the definition of f within the abstract syntax tree (AST) of the module;
  Inspect the set of stack-local variables f.__code__.co_varnames;
  Build a control-flow graph (CFG) for the body of f;
  Analyse the CFG for stack-local names eligible for let-weakening:
  – Elide unreachable CFG nodes;
  – Find all dominated, dominator and immediate dominator nodes;
  – Determine dominance frontiers;
  – Perform live variable analysis;
  – Filter the nodes which can be let-weakened;
  – Annotate assignments in the AST which are let-weakening eligible;
  Walk the AST to instantiate a nest of PythonTerm objects:
  Instantiate PythonLetIn terms for nodes eligible for let-weakening;
  return the PythonTerm corresponding to the body of f;
```

Methods, as opposed to functions, are not reified directly by `reify_syntax`, in order to maintain the identity given in the next section. To reify the definition of some `method`, the underlying `method.__func__` must be reified as `reify_syntax(method.__func__)`, and the bound object `method.__self__` retained, to be supplied as the first argument to the function.

5.4.3 Reification identity

ReiPy's pair of functions `reify_scope` and `reify_syntax` are defined in order to maintain the following identity:

```
function(argument_0, argument_1, ..., argument_n)

≡

reify_syntax(function).body.evaluate(

  reify_scope(function)

    .extend(PythonFunctionLocalScope)

    .declare_locals(

      reify_syntax(function).local_variable_names

    ).assign_locals(
      {
        reify_syntax(function).argument_names[0]: argument_0,
        reify_syntax(function).argument_names[1]: argument_1,
        ...,
        reify_syntax(function).argument_names[n]: argument_n
      }
    )
)
```

The extended scope of this identity is an instance of `PythonFunctionLocalScope`, which represents the scope of a function invocation, with each formal argument name bound to the value of the argument respectively. Since the scope of a Python variable is dependent on the presence or absence of an assignment to its name within the function definition, it is important to declare the names of all local variables at the point the `PythonFunctionLocalScope` is instantiated, so that resolutions against that name which occur before the variable has been assigned result in an `UnboundLocalError` being raised, and do not incorrectly recurse through the lexically surrounding scopes.

This identity would fail to hold for Python bound methods, since there is an implicit first argument value bound to the method object, which does not correspond with an argument position in the invocation expression. Therefore the underlying function object of bound methods must be substituted in their place.

Reliance on this identity is central to the translation strategy implemented by ReiPy. In Section 5.6 we describe how this identity is used to translate the behaviour of Python functions.

5.5 Meta-circular interpreter

In our description of the front-end reifier, we sketched certain aspects of ReiPy's meta-circular interpreter, since the front-end necessarily generates terms acceptable to the interpreter. Our interpreter is *structured* to operate directly upon the AST of the term being evaluated, although the actual operation of the interpreter need not even be implemented for ReiPy to translate terms, as opposed to evaluate terms. This is because the translation of source terms is accomplished by dispatching to code generators *associated* with the interpreter, which do not depend on the interpreter implementation itself. The implementation of the interpreter, if present, simply serves as an operation semantics for the author of the code generator to refer to.

Whilst we could have equally implemented our interpreter as a single function, with a monolithic conditional switch on the type of the evaluated term, we instead chose to implement it in an object-oriented style, with separate classes for each term type.

There are just two super-classes from which every class of the interpreter inherits: `PythonScope` and `PythonTerm`.

The interface for `PythonScope` objects is simply:

```
resolve(self: PythonScope, name: str) -> object
```

As we saw before, the interface for `PythonTerm` objects is:

```
evaluate(self: PythonTerm, scope: PythonScope) -> object
is_immediate(self: PythonTerm) -> bool
immediate_reference(self: PythonTerm) -> object
```

The methods `is_immediate` and `immediate_reference` are used extensively in order to identify terms which always evaluate to the same object, for which that object is known at translation time: `is_immediate` tests if this property holds, and `immediate_reference` is used to extract the reference to that immediate object.

Typically the `evaluate` method of compound terms delegates to the `evaluate` method of its sub-terms. For example a refinement of `PythonTerm` for the `if` and `else` keywords might be implemented as follows:

```
class PythonIfElse(PythonTerm):

  def __init__(self, predicate, consequent, alternative):
    self.predicate = predicate
    self.consequent = consequent
    self.alternative = alternative

  def evaluate(self, scope):
    if self.predicate.evaluate(scope):
      self.consequent.evaluate(scope)
    else:
      self.alternative.evaluate(scope)
```

A class implementing Python name resolution (the use of a local, “nonlocal” or global variable) could be implemented simply as:

```
class PythonLoadName(PythonTerm):

  def __init__(self, name):
    self.name = name

  def evaluate(self, scope):
    return scope.resolve(self.name)
```

ReiPy contains such class definitions for most of the terms in the Python language. Some term types are omitted, such as terms for handling exceptions, since these are as yet unsupported, as we discuss later in Section 5.8.

Similarly each type of scope in Python is implemented as a refinement of `PythonScope`. The most common kinds of scope encountered when using ReiPy are `PythonClosureScope` and `PythonFunctionLocalScope`, which represent the scope over which a function instance closes and the local scope of a function invocation respectively. The construction and implementation of `PythonClosureScope` instances were discussed previously in Section 5.4.1.

`PythonFunctionLocalScope` instances must simply maintain a dictionary of local variable bindings, which might be implemented simply as follows:

```
class PythonFunctionLocalScope(PythonScope):

  def __init__(self, parent=None):
    self.__parent = parent
    self.__declarations = list()
    self.__bindings = dict()

  def declare_locals(self, names):
    for name in names:
      self.__declarations.append(name)

  def assign_locals(self, bindings):
    for name, value in bindings.items():
      if name not in self.__declarations:
        raise RuntimeError('undeclared local: ' + name)
      self.__bindings[name] = value

  def resolve(self, name):
    if name in self.__declarations:
      if name in self.__bindings:
        return self.__bindings[name]
      else:
        raise UnboundLocalError('unbound local: ' + name)
    elif self.__parent is not None:
      return self.__parent.resolve(name)
    else:
      raise RuntimeError('unbound variable: ' + name)
```

Local variables are resolved in the member dictionary of bindings, whereas all other names are resolved by delegation to the parent scope. As discussed above, the list of local names `self.__declarations` is necessary to implement Python's variable hoisting semantics correctly.

ReiPy supports other kinds of scope, notably `PythonImmutableFunctionLocalScope` and also `PythonLetInScope`. The former is a refinement of `PythonFunctionLocalScope` which forbids local variable assignment, and binds argument immutably. Since the resulting scope is stateless, it may be provided lazily during translation, to multiple consumers, without any of the hazards which result from duplicating state. The latter implements the stateless scope underlying our let-in extension to Python, necessary for our let-weakening optimisation.

5.6 Code generation

In the previous sections we have described how Python function objects are reified into instances of `PythonTerm`, accepted by our meta-circular interpreter. Translation of these terms into some target language is achieved by application of our reification identity introduced in Section 5.4.3. In the simplest case of immediate functions which do not mutate their local scope, it specialises to:

```
translate(Target,
  immediate(function),
  [argument_provider_0, argument_provider_1, ..., argument_provider_n],
  mandate
)
≡
```

```
translate(Target,
  immediate(reify_syntax(function).body.evaluate),
  [
    translate(Target,
      immediate(Target, PythonImmutableFunctionLocalScope, mandate),
      [
        immediate(Target, reify_scope(function), mandate),
        immediate(Target, tuple(reify_syntax(function).argument_names), mandate),
        translate(Target,
          immediate(Target, instantiate_tuple, mandate),
          [argument_provider_0, argument_provider_1, ... argument_provider_n],
          mandate
        )
      ],
      mandate
    )
  ],
  mandate
)
```

Informally this identity might be stated as: “the invocation behaviour of the given `function`, translated to some target language, is equivalent to the invocation behaviour of the `evaluate` method of its reified body, when provided an immutable scope which binds its arguments, translated to the same target language”.

The constructor for `PythonImmutableFunctionLocalScope` takes three arguments: the parent scope, a tuple of argument names, and a tuple of argument values. The resulting scope is an extension of the parent, with the argument names immutably bound to the argument values. No assignments are permitted in this immutable scope, but changes to the parent scope are visible from the extension—it is immutable in the sense that it does not instantiate any mutable state, or forward mutation operations to its parent. The convenient (non-standard) function `instantiate_tuple` creates a tuple from its argument list, simply implemented thus: `lambda *arguments: tuple(arguments)`.

If `function` were to possibly mutate its local scope, the translation identity becomes somewhat more complicated, since the scope provider must instead be instantiated non-lazily, which we discuss in Section 5.6.1. ReiPy determines whether a function mutates its local scope, or not, by searching the AST of its reified definition for assignment operations, including `import` statements, and class or function definitions. In certain circumstances let-weakening will transform a function which does mutate its local environment into one which doesn't.

The `translate` operator delegates translation to the code generator attached to the method `reify_syntax(function).body.evaluate`. If the function was constructed using the `def` keyword then its body will be an instance of `PythonBlock`. Alternatively, the body of functions constructed via the `lambda` keyword are instances of some subclass of `PythonExpression`, since syntactically Python's `lambda` constructor does not support blocks. Assuming the body is a block, the code generator for `Target` associated with `PythonBlock` will be invoked. Consider for example that the target language is Python itself—specifically `PythonBlock` rather than `PythonExpression`—then the code generator for `PythonBlock.evaluate` might be implemented as follows:

```python
def translate_python_block_to_python_block(method, argument_providers, mandate):
  self = method.__self__
  scope_provider = argument_providers[0]
  statements = []
  for statement in self.member_statements:
    translated_statement = translate(PythonBlock,
      immediate(PythonBlock, statement.evaluate, mandate),
      [scope_provider],
      mandate
    )
    statements.extend(translated_statement.member_statements)
    if translated_statement.return_expression is not None:
        statements.append(
            PythonExpressionStatement(translated_statement.return_expression)
        )
  if self.return_expression is not None:
    translated_expression = translate(PythonBlock
      immediate(PythonBlock, self.return_expression.evaluate, mandate),
      [scope_provider],
      mandate
    )
    statements.extend(translated_expression.member_statements)
    return PythonBlock(statements, translated_expression.return_expression)
  else:
    return PythonBlock(statements)
```

Note that this code generator is associated not with a Python function instance, but a Python method instance, and so the binding to the object of the method is retrieved using the `__self__` property.

Such a code generator is merely a dispatcher, delegating fragments of the translation to the code generators associated with each statement within the block. The major technicality has to do with ReiPy's handling of resulting values: the abstraction `PythonBlock`, as discussed previously, only supports return statements if they are located at the end of the block. Furthermore the translation strategy employed for translating to `PythonBlock` utilises return statements in the final position to represent the result of the computation, similarly to how our pipeline abstraction of Chapter 4 identifies the component of the pipeline which delivers the result. Consequently there is a certain amount of "plumbing" involved in writing this code generator. This is best understood by example: consider for instance that the following Python function is to be translated to `PythonBlock`.

```
def foo(bar):
  print('Hello')
  return 123 + bar

mandate = {}
translation = translate(PythonBlock,
  immediate(PythonBlock, foo, mandate),
  [ immediate(PythonBlock, 789, mandate) ],
  mandate
)
```

The code generator for `PythonBlock` must first delegate to the code generator associated with `PythonExpressionStatement`, since the first statement in the block is an "expression statement" [Fou21]. This returns, after further delegating to the code generator for `print`:

```
print('Hello')
```

This seemingly trivial translation is only permissible since the code generators for Python builtin functions assume that all builtins are bound under their standard names in the translation target's namespace (this assumption can optionally be controlled via the mandate). Subsequently the code generator associated with `PythonBinaryOperation` is invoked for the addition, which further delegates to the code generator for `PythonImmediate`, for the literal and immediate argument, returning respectively:

```
return 123
```

and

```
return 789
```

Where the return statements denote the result of evaluating each sub-expression. These two fragments are then composed by the code generator for `PythonBinaryOperation`, resulting in the sub-translation:

```
V0 = 123
V1 = 789
return V0 + V1
```

Given the behaviour of `PythonBinaryOperation`'s `evaluate` method, which serves as an operational semantics for implementing its code generator:

```
def evaluate(self, scope):
  if self.operation_kind == 'Add':
    return self.left.evaluate(scope) + self.right.evaluate(scope)
  elif self.operation_kind == 'Sub':
    return self.left.evaluate(scope) { self.right.evaluate(scope)
  elif self.operation_kind == 'Mult':
    return self.left.evaluate(scope) * self.right.evaluate(scope)
  ...
```

The code generator for `PythonBinaryOperation.evaluate` might be implemented as follows:

```
def translate_binary_op_to_python_block(method, argument_providers, mandate):
  self = method.__self__
  scope_provider = argument_providers[0]
  translated_left = translate(PythonBlock,
    immediate(PythonBlock, self.left.evaluate, mandate),
    [scope_provider], mandate)
  translated_right = translate(PythonBlock,
    immediate(PythonBlock, self.right.evaluate, mandate),
    [scope_provider], mandate)
```

```
statements = []
left_name = generate_name()
statements.extend(translated_left.member_statements)
statements.append(
  PythonAssignLocal(left_name, translated_left.return_expression))
right_name = generate_name()
statements.extend(translated_right.member_statements)
statements.append(
  PythonAssignLocal(right_name, translated_right.return_expression))
return PythonBlock(statements,
  PythonBinaryOperation(
    self.operation_kind,
    PythonLoadName(left_name),
    PythonLoadName(right_name)
  )
)
```

This is an example of a fairly typical code generator in ReiPy. The translations of functions invoked by the method are translated by *delegating* to their code generators via `translate`, in this case `self.left.evaluate` and `self.right.evaluate`. The ordering of effects is preserved by collecting translated terms in the order demanded by the operator's semantics, within the list `statements`. Note that intermediate values are bound within the environment of the target fragment under unique names, in this case automatically generated by the function `generate_name`. This is a safe and effective method for maintaining binding hygiene, whereby names from the source fragment are never used to bind values in the target fragments—instead unique names, over the scope of the translation, are always generated whenever a binding is required.

Finally the code generator `translate_python_block_to_python_block` composes these fragments into the final result:

```
print('Hello')
V0 = 123
V1 = 789
return V0 + V1
```

Whilst this example is rather trivial, more complicated examples typically involve interaction with objects other than Python builtins, and the inlining of function implementations, which present additional technical challenges.

5.6.1 Mutable scope

The example code generation process just presented did not involve any mutation to the function's local scope. In the presence of mutable function scope, the scope provided to the translation of the function's body must be instantiated within the translation itself, and bound to a name. This bound name serves as the scope provider, instead of the fragment instantiating the scope, as all users of the scope should have a common view of its state. This is achieved using somewhat more intricate logic, replacing that illustrated in the section above. The effect however is simple—had the scope been mutable in the previous example, the result of the translation would necessarily become:

```
V0 = PythonFunctionLocalScope(
  PythonClosureScope((), (), globals())
)
V0.assign_locals({'bar': 789})
print('Hello')
V1 = 123
V2 = V0.resolve('bar')
return V1 + V2
```

The target fragment instantiates an instance of `PythonFunctionLocalScope`, which is provided to the code generator of the function's body as: `V0`. Here we have assumed not only that `foo` was defined within the global namespace, but also that the translated fragment is intended for execution within that *same* global namespace. If this assumption were untrue, then the invocation of `globals` to obtain the global dictionary would not necessarily obtain the global namespace over which `foo` closed. In general the target fragment must access the source interpreter instance over some kind of channel, which we explore more fully in Section 5.6.5. A simpler yet safe alternative is to raise an error if any binding is accessed on the parent scope:

```
V0 = PythonFunctionLocalScope(PythonUnavailableScope())
V0.assign_locals({'bar': 789})
print('Hello')
V1 = 123
V2 = V0.resolve('bar')
return V1 + V2
```

Here for example the `resolve` method on `PythonUnavailableScope` raises a `RuntimeError`.

5.6.2 Inferring mutability of argument bindings

In general determining scope mutability in Python is somewhat subtle. It is tempting to imagine, given Python lexical scoping rules, that a purely syntactic analysis of a function's definition would suffice to determine which arguments of a function are possibly mutated, and also those which are certainly not. Furthermore, it should be possible to lexically determine the names of all local variables declared (which are not arguments).

The presence of Python's builtin `exec` function does not lead to intractability in this regard, since it does not reflect back assignments to local variables performed by the executed code fragment. What does complicate the analysis however is the possibility of scope reification. Consider the following function:

```
def foo(x):
  bar = lambda: print(x)
  return bar
```

Whilst the argument binding for `x` might never appear to be mutated within this definition, we can not conclude from this fragment alone that `x` is never reassigned. For instance, as we saw in Section 5.4.1, the closure scope of `bar` might be reified elsewhere in the program, and used to mutate the binding:

```
baz = foo(123)
baz()
baz.__closure__[0].cell_contents = 789
baz()
```

Here the first closed-over variable of `baz`, namely `x`, is assigned the value `789`. Running this example renders the following to the console:

```
123
789
```

Furthermore, there are other reflective features in Python, such as stack frame inspection, which could provide others means for binding mutation, which present additional challenges to the analysis. In ReiPy we do not attempt a precise analysis for scope mutation, but instead make a simple approximation based solely upon the syntax of the function definition. In the event that an immutable scope is instantiated by the translated fragment, and then subsequently a mutating operation is performed upon that scope, such that the approximation was erroneous, a fatal exception is raised. This is a safe strategy, in the sense that scope resolution can never return an incorrect result, but introduces an artificial source of runtime errors into the target program.

Prior to Python 3.8—which introduced the `:=` operator—anonymous functions defined by the `lambda` keyword never mutate their local scope. Our syntactical analysis for *named* functions comprises a simple check on every node within the function definition's AST. If the AST does not contains any nodes of the following types:

- `ast.Assign`
- `ast.AugAssign`
- `ast.Import`
- `ast.ImportFrom`
- `ast.FunctionDef`
- `ast.ClassDef`
- `ast.AsyncFunctionDef`

then ReiPy's analysis concludes that the function does not mutate its local scope. In the exceptional case that a function does mutate its local scope and does not contain any of these nodes in its AST—for example if the local scope is mutated only via reflection—it may be annotated with a `@mutable_local_scope` decorator in order to override the outcome of the analysis on a case-by-case basis.

Our current implementation of ReiPy performs its *let-weakening analysis* before scope mutation analysis, but the *let-weakening transformation* subsequent to scope mutation analysis. This is for purely incidental reasons, and not due to any true dependency. Hence AST nodes which mutate only variables within the set of let-weak names are excluded from this syntactical analysis, since their contribution to scope mutation will be removed during the let-weakening transformation. This is important, since otherwise the vast majority of function definitions in any practical program would lead to unnecessary scope materialisation in their translated form, and nullify many opportunities for constant propagation and folding.

5.6.3 Implementing `translate`

There is considerable design freedom when it comes to implementing the `translate` operator in Python. ReiPy takes advantage of the ability to dynamically assign new attributes to Python function instances. If the provider of the function being translated is immediate, then the presence of an attribute on the function object with a “well-known name” is dynamically checked. The current version of ReiPy associates code generators for some `target` language with an attribute based on the target language name: `'_translate_invoke_to_' + target.name`. If an attribute with that names is present on the function, it is used to determine the associated code generator. Translation is delegated to that generator, and if it produces a result then that translation is preferentially returned. If however the code generator opts to return no result, or the function is not immediate and hence no preferential generator is resolvable, then `translate` defaults to delegating the task of translation to the `default_translate_invoke` method of the target language. This is illustrated within the following implementation of `translate`:

```
def translate(target, function_provider, argument_providers, mandate):
  assert isinstance(function_provider, target)
  for argument_provider in argument_providers:
    assert isinstance(argument_provider, target)
  if function_provider.is_immediate():
    function = function_provider.immediate_reference()
    generator_attribute = '_translate_invoke_to_' + target.name
    if hasattr(function, generator_attribute):
      generator = getattr(function, generator_attribute)
      result = generator(function, argument_providers, mandate)
      if result is not None:
        assert isinstance(result, target)
        return result
  result = target.default_translate_invoke(function_provider,
    argument_providers, mandate)
  assert isinstance(result, target)
  return result
```

Clearly this particular implementation of `translate`, and specifically its delegation policy, is arbitrary. There are many other possible designs which would conform to the same interface, and render semantically acceptable translations. Additional cases could be inserted into the dispatch policy, for example non-immediate providers could associate code generators against other well-known named attributes attached to the provider itself, and be delegated to preferentially over the default translation.

The default code generator for a given target can be implemented in any manner whatsoever. In all of our experiments we designed our default code generator to perform code generation by means of reification, as detailed earlier in this section.

5.6.4 Contextual generation

All of our previous examples have involved an extreme case regarding the information that is known about the function under translation: either the translated function was immediate and thus translatable; or unknown and so untranslatable; or some value was either immediate and thus inlined; or unknown and so its resolution was delayed.

Many scenarios in which late-bound code generation is applicable involve cases which are less binary, where a certain degree of information about an object or value is available at translation time, but the material instance is not.

A scenario of this kind occurs when translating the behaviour of a function by means of reification: even if the function being translated itself is immediate, the local scope within which the syntax of the function definition is evaluated is in general *not*. The translated term must generally instantiate some representation of the *state* of the scope, such as a stack frame or heap allocated dictionary, dynamically for each future effective invocation of the behaviour. Under certain conditions, such as scope immutability, this instantiation may be elided, but it is necessary in general to allocate dynamic temporary storage for the function's evaluation state. Hence although this local scope is neither generally immediate, nor generally possible to materialise at translation time, this does not imply every conceivable property of the scope instance is unknown.

In certain cases some ancestor of the scope might be immediate, for example if the Python function being translated is immediate, then the parent of the local scope is certainly immediate, since it is the function's closure scope. Alternatively, if the translated function is invoked with a literal argument (giving rise to an immediate argument provider) and the corresponding argument variable is never mutated, then the resolution of that argument's formal name in the local scope is also immediate at translation time.

If it were devoid of any mechanism to encode such information, ReiPy would necessarily fail to produce useful or performant translations when encountering such scenarios. Furthermore, in our experience applying ReiPy to certain practical problems, such as those encountered in Chapter 4, such scenarios are relatively common, if not representing the majority of cases. ReiPy's solves this problem using *contextual generators*.

A contextual generator is simply a code generator which is valid only within some specific sub-context of the translation process: the correctness of the generated code depends on certain assumptions about the context in which it is executed, which do not hold more widely throughout the translated program. Hence contextual generators are typically instantiated dynamically during translation, applied selectively within their domain of validity, and then discarded afterwards. For example, in order to represent the fact that the parent scope of a non-immediate local scope is immediate, a contextual generator which provides that immediate closure scope is constructed for the requisite language target. This contextual generator is then associated with the *provider* of the local scope, in such a way as to override the code generated when translating an invocation of the `parent` method on the provided local scope. We call this the *immaterial scope representation* problem, or ISR problem.

One solution to the ISR problem is to override the code generator for loading the `resolve` and `parent` attributes on the provided scope. If these methods are declared constant (in the sense that the method *implementation* may not be changed after being defined) the default code generator for loading their named attribute would return the bound method object as an immediate value in the target language. In turn, the code generator associated with the bound method object is called upon to generate an implementation of the behaviour of the method, valid for any `self` instance provided. Using a contextual code generator, it is possible to override the *former* code generator, representing a specialised method object in place of the original bound method object. The code generator of this specialised method object only generates code for representing some *specific* closure scope, and is therefore only valid for a particular binding of `self`.

Consider using an attribute with some well-known name, to associate our contextual code generator with the local scope provider: `'_override_translate_load_attribute_to_' + target.name`. The code generator associated with our metacircular evaluator's operator for loading attributes, upon checking for this attribute on the object provider, can therefore delegate preferentially to the overriding generator. If the overriding code generator provides a specialised implementation of `resolve` when accessing the `'resolve'` attribute of the scope, associated in turn with a code generator which provides immediate values for bindings known at translation time, then the ISR problem has been solved, in part. This solution can be extended, by similarly overriding the code generator for the `parent` method the scope provider can also represent ancestry information about the future scope. Note that this solution relies heavily on the metacircularity of ReiPy's interpreter, whereby code generators for translating source terms stage the interpreter itself, dynamically dispatching to code generators for operations *internal* to the interpreter, such as operations on scopes, in addition to operations which appear literally in the source term.

This is just one potential application of contextual generators to solving the ISR problem. Another solution might associate the contextual generator with a class dynamically instantiated duration translation. With this approach, code generation is selectively specialised by extending the class of certain values within a particular context. This specialised class is automatically instantiated during the translation process, and used to override the code generation of operations only within fragments on the translated term for which the specialisation in valid. This is quite a general technique for representing binding at translation time, whereby immediate values are bound statically to the class extension, and unknown values remain as variables of the class instance. This was the approach taken to solve the ISR problem in our rendering experiment of Chapter 4. Classes with static bindings for their scope ancestry and any constant values were instantiated during translation, in order to represent partial information regarding scopes.

A hazard inherent to instantiating contextual code generators, is that the specialised generator might be called upon to generate code situated outside of the region to which the specialisation applies. The semantics ReiPy requires of, but does not enforce upon, provider objects are intended to mitigate this hazard. If contextual code generators are (i) only ever associated directly with providers, are (ii) defined to be valid wherever the provider is valid, and (iii) are never called upon to generate code outside the provider's valid context, then the contextual code generated must be valid. Therefore it remains to define the validity rules for providers.

ReiPy's semantics for providers are as follows:

- The provider must be valid within the context in which the generated code fragment is executed.
- The provider terms supplied to a code generator may be incorporated into the translated code fragment zero times, once or several times.
- Providers may be reused as providers to code generators which are delegated to form *sub-fragments* of the calling code generator's fragment, so long as the provider remains valid within the context of the sub-fragment.
- References must not be established to any provider which could outlive the invocation of the code generator. Specifically references to providers must not be directly or indirectly attached as metadata to the *returned* fragment, since the provider is only necessarily valid *within* the context of the fragment.
- Providers must be idempotent upon execution, and must not invalidate any other provider supplied to the code generator.

If providers are constructed and used according to these rules, then not only should the correct value be provided within the translated fragment, but any contextual code generators associated with the provider should only ever be available for delegation within the context of their speciality.

An intuitive way to think about the semantics imposed upon providers and contextual code generators, is that ReiPy generates code by means of mutually recursive dynamic dispatch between many fragmentary code generators. The call stack of the translation process mirrors the nesting of sub-terms within the translated term—code generators recursively call other code generators to translate sub-terms of the result, which are combined together by the calling code generator. A particular code generator might specialise a generator or provider to the context of the sub-term it is currently generating, a specialisation which is valid throughout the downward sub-stack of code generators it delegates to. In general this kind of specialisation is invalid in the upward sub-stack of calling code generators, and so any reachable reference which outlives the invocation of the code generator is potentially hazardous. This has an informal analogy to the downward and upward "FUNARG" problem [Mos70]: providers and contextual code generators are in general only valid when eligible for stack-based allocation (*downward*) and pose a hazard when heap-allocated (*upward*). Python does not offer any means for requesting stack-allocation of objects, which would assist in formally enforcing these semantics.

5.6.5 Object persistence

During translation any immediate and immutable values encountered during translation, which are necessarily referenced by the translated term, can simply be rematerialised within the translated term. For example, numeric literals are usually represented as numeric literals or bit-strings throughout the result of the translation. If however the translation process encounters a non-immediate reference, or an immediate reference to a mutable object, the translated term must instead be generated with free variables which are subsequently linked to the requisite object upon execution. We term this problem "object persistence", in analogy to the issue of "cross-stage persistence" in multi-stage programming languages.

In general ReiPy supports object persistence by means of the translation mandate. When persistent objects are encountered during translation, a (translation-wide) unique term within the target language is allocated to represent references to that object. This can be performed in a target specific manner: for example when targeting Python itself the same function `generate_name` might be used, as employed when generating names for binding intermediate results—but in this case the code for binding the name itself is not initially generated. It is crucial that the association between the uniquely allocated term and the persistent object is retained, as this association is necessary for subsequently linking the translated term prior to execution.

A convenient way to retain these associations is via the mandate object. A directory of all persistent objects, associated as an attribute of the mandate, is populated whenever a new persistent object is encountered, registering the unique term allocated along with some procedure to obtain the persistent object itself. Whilst this directory could instead be associated with fragments of the translation, associating it with the mandate is more convenient for two reasons. Firstly, with the code generator attached to the mandate—an object shared between all code generators contributing to a translation—there is an opportunity to cache and reuse the allocated term for the persistent object, amongst its several separate uses. Secondly, if the directory were attached to individual code fragments, it would become necessary for code generators to merge the directories of persistent objects, whenever code fragments are combined. Although this is a relatively simple problem to solve, it introduces unnecessary complexity to the implementation of every ReiPy code generator.

The directory of terms allocated for persistent objects can be inspected, after the eventual translation is formed, in order to prune any unneeded terms. ReiPy typically engenders the generation of many code fragments during translation which are subsequently elided from the translation result. For instance these elided fragments might include providers for unused arguments, or generic fragments which are eliminated by opportunistic inlining or constant folding optimisations. Any persistent objects which are determined to be ultimately unused can simply be removed from the directory.

In order to execute any translation which includes references to persistent objects, the execution environment must first be prepared, ensuring that those terms allocated to each persistent object resolve correctly to the requisite object at runtime. This is analogous to the "linking" step which traditionally precedes execution for ahead-of-time compiled languages. In general the persistent objects referred to are not necessarily immediate at linkage time either, and thus a bidirectional channel must be established to allow for runtime resolution of the object. For example the source program might refer to a mutable variable, addressed by name, in the originating process' global scope. Each time such an object is dereferenced by the translation during execution, messages must be exchanges over this channel with the originating processes, in order to obtain information about the current state of the global binding.

5.7 Offloading computation to the cloud

In Section 5.2 we described how ReiPy can be used to offload computation across a network—for instance offloading the computation of a hash over database values, to the same machine as is hosting that database. This particular example was primarily motivated by the pragmatic concern of minimising network bandwidth consumption, although the total execution time would likely be reduced also.

A similar situation applies to resources which are hosted in "the cloud", whereby databases and other stateful or stateless services are managed as a subscription service, physically co-located in some data center. Our motivating example is also applicable when considering the heterogeneity of network bandwidth, internal and external to such a data center. Rather than offloading computation to the very same machine as is hosting the database, which strictly minimises network bandwidth consumption, computation might instead by offloaded to a machine co-located with the database. If this machine is attached to the same internal data centre network as the database server, since that local network's available bandwidth is likely greater than that to the client machine, the same pragmatic advantage is gained.

5.7.1 Basic scenario

In this section we return to the predicate function introduced in Section 5.2, and show in detail how it is translated into a form suitable for remote execution. Subsequently in sections

5.7.2 and 5.7.3 we will adapt this basic scenario, incorporating specific considerations for cloud execution.

Consider again the definition of `predicate`, whose computation we desire to offload:

```python
def predicate(some_key, some_hash):
  value = DB.query('GET KEY ' + some_key)
  hash_result = HASH(value)
  return hash_result == some_hash
```

Throughout we will assume that `predicate` was instantiated within the following context:

```python
def build_predicate(DB):
  HASH = DB.get_configured_hash_function()
  def predicate(some_key, some_hash):
    value = DB.query('GET KEY ' + some_key)
    hash_result = HASH(value)
    return hash_result == some_hash
  return predicate

database_connection = RemoteDatabaseConnection(
    'MODE=tcp;HOST=database.example.com'
)
predicate = build_predicate(database_connection)
```

Note that `DB` is an argument to the function lexically enclosing `predicate`, the value of which is instantiated before `build_predicate` is invoked. However, `HASH` is bound to a function determined by—potentially effectful—communication with the database.

By applying the `translate` operator to our function closure, we seek to obtain a transformed representation of the *invocation behaviour* of `predicate`. This transformed representation must only comprise Python terms common to both the local and remote interpreter evaluation scopes, specifically eliminating any use of the names `DB` and `HASH`, which in our scenario are not assumed to be common terms. We apply ReiPy's `translate` operator to obtain our translation:

```
interesting_hash = 0x276104A8
translation = translate(PythonBlock,
  immediate(PythonBlock, predicate, mandate), [
    PythonBlock([], PythonLoadName('argument_0')),
    immediate(PythonBlock, interesting_hash, mandate)
  ], mandate
)
```

The very first step of the transformation occurs within ReiPy's preprocessor. The control-flow graph for `predicate` is generated, according to the process we described in Section 5.3, the result of which is illustrated in Figure 5.1. Applying our let-weakening algorithm to this control-flow graph yields the following transformed definition:

```
def predicate(some_key, some_hash):
  let UniqueName0 = DB.query('GET KEY ' + some_key) in:
    let UniqueName1 = HASH(UniqueName0) in:
      return UniqueName1 == some_hash
```

Both `value` and `hash_result` have been let-weakened. They have been renamed at both their definitions and uses to uniquely generated names, not occurring elsewhere within the source fragment. Given the binding structure manifest within this let-weakened form, but without any domain-specific customisation, ReiPy would produce the following translation:

```
V0 = OriginProxy(0x7efd5bf41d38, ctypes.py_object).query
V1 = 'GET KEY '
V2 = V1 + argument_0
```

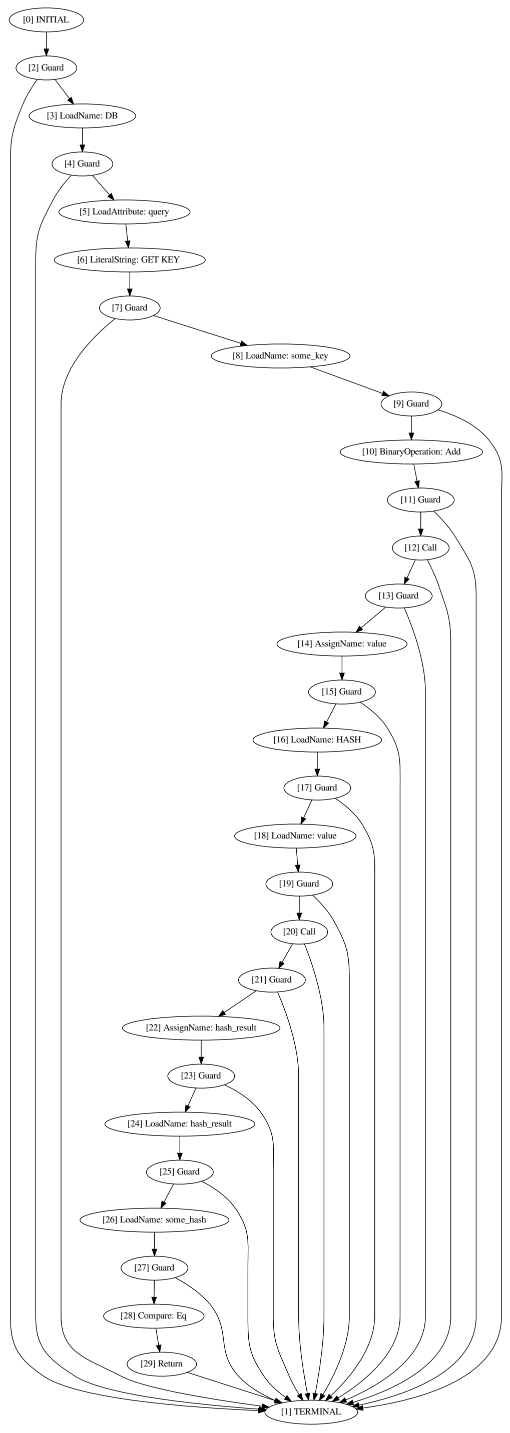

Figure 5.1: Control-flow graph corresponding to the function definition of `predicate`. All of the edges landing on the terminal node, except the single edge from the `return` node, correspond to exceptional control flow paths. Ignoring these exceptional edges, this is a linear graph, since `predicate` does not contain any conditional statements, or any loop constructs.

```
V3 = V0(V2)

V4 = OriginProxy(0x7efd5bf7cf78, ctypes.py_object)

V5 = V4(V3)

V6 = 660669608

return V5 == V6
```

The bound value for `DB` is represented in the translation as a proxy. `OriginProxy` represents an object in the originating interpreter's address space, here parameterised by the virtual address of the instance on the heap, along with its primitive type. This proxy forwards all object interactions over the network connecting the original and target interpreters, and so performs the query using double network indirection: first the query string would be sent back to the original interpreter, and then forwarded to the database, the database would then return the query result back to the original interpreter, which finally forwards it to the target interpreter. This would not reduce our network bandwidth consumption, but only exacerbate it.

A similar issue applies to the binding of `HASH`, however this only involves a single level of network indirection, computing the hash within the originating interpreter, yet again exacerbating our bandwidth consumption. Clearly, in order to minimise our network bandwidth usage, these two proxies must be eliminated from the translation. We will first demonstrate how the proxy for `HASH` is removed, and then subsequently the proxy for `DB`.

Before we proceed, one might object to the fact that ReiPy has inlined the virtual address of the object *currently* referenced by `DB` and `HASH` into the translation, rather than emitting a call to `resolve('DB')` and `resolve('HASH')` on an `OriginProxy` for `predicate`'s scope. ReiPy was enabled to inline these references, since we are using it in a configuration whereby all names starting with a capital letter are assumed to be declared constant. This aligns well with Python's standard naming conventions, and reduces the need to explicitly and tediously declare the constancy of various bindings to ReiPy. If `DB` was instead named `db`, the default translation would be generated as follows:

```
V0 = PythonImmutableFunctionLocalScope.resolve(
  PythonImmutableFunctionLocalScope(
    OriginProxy(0x7fedd27fa128, ctypes.py_object),
    ('some_key', 'some_hash'),
    (argument_0, 660669608)
  ),
  'DB'
).query
V1 = 'GET KEY '
V2 = V1 + argument_0
V3 = V0(V2)
V4 = zlib.adler32
V5 = V4(V3)
V6 = 660669608
V5 == V6
```

To eliminate the proxy for our hash function, such that the hash can be computed on the remote machine, we must override the proxy generator corresponding to the hash function used. Given that which hash function is applied depends on the runtime result of the method `get_configured_hash_function`, we must override the proxy generator corresponding to *all potentially used* hash functions. If we assume that the function is only ever bound to those hash functions available in Python's `zlib` module, we can override each of their generators thus:

```
for hash_function in [crc32, adler32]:
  @proxy_generator(PythonBlock, hash_function)
  def proxy(reference, mandate):
    return PythonBlock([],
      PythonAttributeLoad(PythonLoadName('zlib'), hash_function.__name__)
    )
```

Here we apply ReiPy's `@proxy_generator` decorator to each hash function in `zlib`, overriding its representation in the target language `PythonBlock`. This generator returns the terms "zlib.crc32" and "zlib.adler32" for representing the CRC32 and Adler-32 hash functions respectively. Due to a technical difficulty, it is not possible to apply an override directly to a builtin Python function, since ReiPy internally attaches the generator to the function using an attribute, but builtin Python objects do not support dynamically settable attributes. Therefore we must define and use a trivially wrapped version of these builtin functions:

```
def crc32(value):
  return zlib.crc32(value)


def adler32(value):
  return zlib.adler32(value)
```

With our proxy generator overrides now associated with their hash functions, the resulting translation becomes:

```
V0 = OriginProxy(0x7efd5bf41d38, ctypes.py_object).query
V1 = 'GET KEY '
V2 = V1 + argument_0
V3 = V0(V2)
V4 = zlib.adler32
V5 = V4(V3)
V6 = 660669608
return V5 == V6
```

We have declared the availability and meaning of the name 'zlib' in the target scope, by means of overriding the relevant proxy generator. The substitution of this hash proxy was legitimate, since it corresponds to a *pure function*. Now all that remains is to remove the proxy corresponding to the database connection object. We will first demonstrate how the proxy may be rewritten from a TCP/IP connection to a local Unix domain socket connection. Then subsequently we describe a variation whereby a TCP/IP connection is retained, but using a rewritten connection string.

Similarly to how we eliminated the hash function proxy, we will override the proxy generator for our connection object: however in this case, for the sake of generality, we will implement a parametric generator, which is parameterised by the mandate. This will allow us to modify aspects of the connection's representation without modifying the generator definition itself:

```
@proxy_generator(PythonBlock, database_connection)
def proxy_connection(reference, mandate):
  try:
    local_connection_string = mandate.remote_to_local_database_map[
      reference.connection_string
    ]
    proxy_name = generate_name()
    return PythonBlock(
      [PythonAssignLocal(proxy_name,
        PythonCall(PythonLoadName('LocalDatabaseConnection'),
          [PythonImmediate(local_connection_string)]
        )
      )],
      PythonLoadName(proxy_name)
    )
  except KeyError:
    pass
```

This generator dynamically checks, at translation time, whether the mandate contains a mapping for the object's connection string. If a mapping exists then the representation of the connection is overridden, otherwise the generated representation falls back to the default proxy. For reasons of exposition, we implemented this generator to emit code including an assignment statement—the bound name is generated using `generate_name`, in order to maintain hygiene. The code generated estantiates an instance of the class bound to `LocalDatabaseConnection` in the target scope, using the remapped connection string. This instance is then bound to the uniquely generated name, and the resulting representation corresponds to the value resolved by that generated name.

In order to make use of this generator override, we must first populate our mandate with a suitable connection mapping. Here we accomplish this by defining a custom class for our mandate, which populates the requisite `remote_to_local_database_map` attribute:

```
class MyMandate:

  def __init__(self):
    self.remote_to_local_database_map = {}

  def map_remote_to_local_database(self, remote_string, local_string):
    self.remote_to_local_database_map[remote_string] = local_string
```

Then to make use of our connection string mapping feature, we simply instantiate, populate and supply our custom mandate to the `translate` operator:

```
interesting_hash = 0x276104A8
mandate = MyMandate()
mandate.map_remote_to_local_database(
  'MODE=tcp;HOST=database.example.com', 'MODE=uds;PATH=/tmp/database.sock'
)
translation = translate(PythonBlock,
  immediate(PythonBlock, predicate, mandate), [
    PythonBlock([], PythonLoadName('argument_0')),
    immediate(PythonBlock, interesting_hash, mandate)
  ], mandate
)
```

Hence we have eliminated all proxies from the resulting translation, arriving at the result we originally sought in Section 5.2:

```
V0 = LocalDatabaseConnection('MODE=uds;PATH=/tmp/database.sock')

V1 = V0.query

V2 = 'GET KEY '

V3 = V2 + argument_0

V4 = V1(V3)

V5 = zlib.adler32

V6 = V5(V4)

V7 = 660669608

return V6 == V7
```

5.7.2 Hostname rewriting variation

In a cloud computing context, where our execution target is likely to be a *neighbouring* machine of the database server, rather than the same machine, it might be more appropriate to rewrite only the TCP/IP hostname of the connection string—although this might be in some cases unnecessary, depending on the network routing configuration of the data centre. This type of rewrite is achieved in an straightforward and analogous manner to our domain socket substitution, via a novel attribute of the mandate:

```
mandate.map_hostname(
  'database-node-42.cloud.example.com',
  'database-node-42.local'
)
```

Any remote connection bound to `'database-node-42.cloud.example.com'` within the translation will hence be represented as follows:

```
V0 = RemoteDatabaseConnection('MODE=tcp;HOST=database-node-42.local')
...
```

In contrast to the proxy generators for our hash function, the validity of these connection substitutions does not follow from the functional purity of the connection object. Clearly, results obtained from database connection objects inherently depend on the mutable state of the database. What we are asserting however, is that the extent of *impurity* within the connection extends only to its dependence on the global database state, and no further. Therefore it is valid to substitute another connection, so long as it connects to the *same* database.

5.7.3 Issuing computation for remote parallel execution

Finally we must consider how our translated fragment is offloaded, for remote execution. Suppose that there are three database keys we wish to check against our static hash value. This could be issued in parallel as follows:

```
results = {}
for key in ['data_123', 'data_456', 'data_789']:
  results[key] = execute_remote(
    database_connection.get_proximate_host(),
    translation, {'argument_0': key}
  )
for key in results:
  result = await results[key]
  print(key + ' : ' + str(result))
```

Execution of our `translation` is equivalent to applying `predicate` to `key` and `interesting_hash`, however it occurs on the remote server returned by the method `get_proximate_host`. This method is free to return any available server co-located with the database, perhaps supplying results according to some adaptive, or simple round-robin policy. The function `execute_remote` takes a dictionary of bindings to serialise, and make available within the remote scope, and returns an “awaitable” object, which represents the eventual result of the computation. After issuing all of the keys for parallel execution, the results are awaited and then printed to the console, for example rendering:

```
data_123 : False
data_456 : True
data_789 : False
```

Prior to execution, the remote machine must populate the implicitly assumed bindings to `zlib`, and the database connection classes within its execution scope, along with the deserialised binding for `argument_0`:

```
scope = {
  'zlib': zlib,
  'LocalDatabaseConnection': LocalDatabaseConnection,
  'RemoteDatabaseConnection': RemoteDatabaseConnection,
  **{name: deserialised_values[name]
    for name in formal_names}
  } # e.g. {'argument_0': 'data_123'}
exec(code, scope)
```

It is interesting to consider one final variation to this scenario, in which the translation is specialised to each key separately. Although in our particular context this is unlikely to be of any pragmatic benefit, it illustrates the general possibility of optimising the translated fragment:

```
results = {}
for key in ['data_123', 'data_456', 'data_789']:
  translation = translate(PythonBlock,
    immediate(PythonBlock, predicate, mandate), [
      immediate(PythonBlock, key, mandate),
      immediate(PythonBlock, interesting_hash, mandate)
    ], mandate
  )
  results[key] = execute_remote(
    database_connection.get_host(), translation, {}
  )
```

Given that `key` is here provided as an immediate value, ReiPy may constant fold the query string at translation time. The following optimised translation is generated for the key `'data_123'`:

```
V0 = LocalDatabaseConnection('MODE=uds;PATH=/tmp/database.sock')
V1 = V0.query
V2 = V1('GET KEY data_123')
V3 = zlib.adler32
V4 = V3(V2)
V5 = 660669608
return V4 == V5
```

5.8 Conclusion

In this chapter we presented ReiPy, a Python library for performing late-bound code generation. Its principal operator `translate` acts upon (providers for) function instances, translating their invocation behaviour to syntactic closures in some target language. Although the only target language we explicitly considered was Python itself, other target languages are admissible by design. ReiPy consists of four parts: (i) a front-end preprocessor; (ii) a reifier; (iii) a meta-circular interpreter; and (iv) a back-end collection of code generators.

The primary responsibility of the preprocessor is to let-weaken program fragments, a syntax-based transformation intended to solve the problem of propagating information pertaining to named variables between modular code generating units. Let-weakening conservatively replaces certain assignment statements to mutable named variables with lexically-scoped and renamed immutable bindings (let-in expressions). Our implementation of let-weakening is based upon a traditional SSA transformation, and pruned according to a supplementary live variable analysis. The definition of let-weakened names thus becomes apparent to code generators acting upon sub-terms within its scope, since the definition of the name is now unambiguously resolvable within the scope hierarchy. Without let-weakening, information that pertains to a variable definition would have no mechanism for propagating from definition to use during translation—for instance preventing all constant-propagation based optimisations.

Our reifier for Python function instances supports the reification of both: (i) the syntax of the function definition; and (ii) its closure scope. It builds upon CPython's underlying facilities for introspection, since the Python language itself lacks an implementation-independent definition of the requisite introspection operators. The resulting reified structures conform to the interfaces accepted by our metacircular interpreter. ReiPy's metacircular interpreter is defined as a collection of independent, yet compatible classes—as opposed to a monolithic evaluation function. Individual classes implement the semantics of different terms within the Python language, and its different scopes.

Code generators for a particular target language are associated with methods of the metacircular interpreter, and also optionally with functions invoked directly or indirectly by the translated function itself, which override those of the interpreter. Translation proceeds recursively by dynamic dispatch between these modular code generating units, and under a particular mandate. The mandate of the translation includes information pertaining to the translation unit as a whole, for example whether bindings to Python's builtin functions may be considered immutable, or not. Side-effects within the translated function are permitted, and their ordering is preserved within the translation. Our code generation system heuristically determines the immutability of certain function local scopes, in order to minimise the allocation of mutable state within the resulting translation, however the outcome of this heuristic may be overridden using an appropriate function decorator. In those cases where a separable translation cannot be attained, we described ReiPy's approach to object persistence.

We introduced the concept of contextual generation, whereby specialised code generators are instantiated during the translation process by one code generator, and dynamically provided to and for the use of another code generator. Contextual generation was applied to solve the immaterial scope representation problem, which occurs when a particular scope is not fully immediate at translation time, but must be represented (in some non-trivial way) in order to effectively proceed with the translation. This problem typically occurs when representing the local scope of a function which is invoked by the translated function—even if the invoked function itself is immediate, its local scope is not, since the local scope is in general only materialised at the point of each invocation. Contextual generation allows such kinds of immaterial scope to be represented, whereby information about the scope and its ancestors is maintained and propagated during the translation process.

Given that ReiPy is more a framework than a concrete implementation system, its capabilities and limitations depend strongly upon the capabilities and limitations of the code generators made available to it. Whilst as described ReiPy is capable of translating side-effecting functions, it has certain notable limitations within its current design and implementation. For instance it does not currently support: (i) early function return; (ii) throwing or catching exceptions; (iii) coroutines or generators; (iv) variable shadowing; or (v) non-positional arguments. Consequently, most existing Python programs are not yet handled by our system. Supporting any or all of these features would require redesigning and reimplementing a substantial portion of ReiPy.

Overcoming the first three of these limitations might be accomplished by substituting the existing direct style metacircular interpreter for a continuation-passing style interpreter—while leaving the interpreted program and code generators in direct style. Some rudimentary experiments along these lines have already shown promising results, but extensive further work would be required to confirm that this approach is feasible. Notably, this suggests that late-bound code generation is not inherently limited to functions exhibiting only local control flow, and the limitation is instead an artefact of the style in which our metacircular interpreter was written.

Support for variable shadowing could in principle be accommodated by modifying the preprocessor, such that names which shadow variables are renamed in the same manner that let-weakened bindings are renamed: again simplifying the task of the code generation system by eliding certain complexities via a syntactic transformation.

To support non-positional arguments would require a more substantial modification of the late-bound code generation interface itself. The function signature of `translate`, and of its code generators, both feature a list of argument providers. Support for optional named arguments, and also variable arguments, would require this list to be substituted for a different data structure, for example some kind of argument collection object. Alternatively the list of "argument providers" could be replaced by a single "arguments provider", which provides a complex object containing all of the arguments, named, positional or variable. Object-oriented languages such as Python typically adopt a convention that the subject of the invocation is simply the first positional argument, frequently named `self`, which suggests a more general interface for `translate` with simply three parameters: (i) the target language; (ii) a single complex-arguments provider; and (iii) the mandate.

Despite ReiPy's significant limitations, there are reasons to believe it has some specific utility in its current design, although further work would be required to implement target languages other than Python, even so. In those cases where it is desired to offload some fragment of a program to a hardware accelerator, the ability of that accelerator to outperform the host processor, either in terms of speed of power efficiency, is often predicated on its specialisation. The accelerator is only capable of (efficiently) executing programs which conform to some specialised set of restrictions, which it leverages to improve its performance. For example such kinds of restriction often limit accelerated fragments to be absent of non-local control flow. Therefore, in particular situations, the limitations of ReiPy are well-aligned with other external constraints, such that the programs it handles span a useful set.

Chapter 6

Conclusion

6.1 Summary

The primary contribution of this thesis is in its definition, and demonstration, of *late-bound code generation*: a novel technique for enabling computer programs to affect pragmatic control over their own execution. Late-bound code generation is based upon syntactification, an approach to translation, which represents the translated fragment as a syntactic closure. Syntactification proceeds from a trivial proxy-based seed translation, which is then progressively elaborated upon by a process that materialises information implicit within its scope into syntactical terms. Within late-bound code generation, as defined in this thesis, each elaboration step is performed by a modular code generator at runtime, which is selected by dynamic dispatch.

We described the principal operator of late-bound code generation: `translate`. Our operator acts upon (providers for) potentially side-effecting function instances, returning a translation of its behaviour in the given target language. Generating translations using `translate` does not require the programmer to resort to programming in a multi-level language, nor does it interfere with the interfaces at which software abstractions are composed. We defined its operand requirements, including the translation mandate, and function and argument providers, the latter of which are also syntactic closures in the target language.

An application of late-bound code generation was illustrated, by applying it to the problem of offloading the rasterisation of a smartphone user interface to an FPGA. We first defined our rendering architecture, comprising a statically scheduled pipeline, suitable for specialising to a particular application, and amenable to place-and-route style optimisation. We then demonstrated how late-bound code generation can be applied to generate an instance of this architecture, specialised to a smartphone application written in terms of contemporary abstractions, and featuring both standard and programmatically defined user interface components. To generate an efficient, statically scheduled pipeline it was necessary to supplement the application with a number of annotations, which we described and justified. We detailed the mechanism by which the state of the pipeline was synchronised with the state of the application, between each frame. The power consumed by our pipeline, when executed on an FPGA, was estimated, although we were unable to directly compare this result with a baseline power consumption figure for a commercial smartphone.

Finally we described ReiPy, a library for performing late-bound code generation within the Python programming language. We offered a solution to the problem of propagating information pertaining to variable assignments between modular code generators, based on a preprocessor transformation called let-weakening: replacing assignments with lexically scoped immutable bindings where possible. Late-bound code generation gives rise to an issue termed the "immaterial scope representation problem". Our solution to this problem, which is based on the more general technique of contextual generation, was outlined. We assessed the limitations of ReiPy as currently defined.

6.2 Critique

Late-bound code generation is both a technique and a mechanism, in the same way that dynamic dispatch is a both a technique and a mechanism. Generically it refers to a set of related mechanisms, which all share a particular character, along with usage patterns for applying those mechanisms to specific programming problems.

Similarly to how one mechanism for dynamic dispatch differs from another with respect to the priority list of its *dispatch policy* when a method is invoked—for instance one mechanism might support multiple inheritance whereas the other might not—different mechanisms for late-bound code generation also differ in their priority list for dispatching to code generators during translation. In this thesis we attempted to define both the general character of the technique, a basic implementation of a particular mechanism, and also explore certain variations on that basic mechanism, for instance contextual generation.

Despite these variations among the various potential mechanisms for late-bound code generation, the technique does in general commit to a fixed front- and back-end interface. Specifically the front-end interface for requesting the translation of a function:

```
Target translate(
  Type Target,
  Target function_provider,
  List<Target> argument_providers,
  ⊤ mandate
),
```

and the back-end interface for supplying translation fragments, implemented by individual code generators:

```
Target generator(
  Function function,
  List<Target> argument_providers,
  ⊤ mandate
).
```

This particular pair of interfaces, and their (informal) semantics in combination with general techniques and patterns for their application, comprise our definition of late-bound code generation—and it is upon this basis that we organise our reflections on this thesis.

Whilst various deficiencies might be identified in a particular implementation of one mechanism for late-bound code generation, those deficiencies do not necessarily undermine the validity of the technique in general, just as the absence of parametric polymorphism in early versions of Java's type system did not undermine the validity of object-orientation as a technique. In contrast however, any deficiency of late-bound code generation in supporting pragmatic control, which is *entailed* by this abstract definition does indicate a serious flaw.

Finally, the *tractability* of implementing the technique in general is directly relevant. For example if the technique creates implementation challenges which are (i) impractical to overcome, (ii) do not readily permit implementations which scale well, or (iii) places undesirable requirements on its users, then these are also serious deficiencies in the approach.

In the following sections we reflect upon late-bound code generation, formulating our criticism according to these terms. Subsequently in Section 6.4 we return to speculate upon further work and future research directions, which were left unexplored within this thesis.

6.2.1 Interface stability and generality

Our front-end and back-end interfaces for late-bound code generation evolved slightly during our experimentation with the technique. However, somewhat to our surprise, the relatively simple interfaces listed above, if applied retrospectively, suffice for all the pragmatic control tasks upon which we embarked. They serve equally for the code generators associated with application-level functions, as the code generators associated with each operator of our meta-circular evaluator.

This is in part due to the extensibility of the data types used. We came by this extensibility by virtue of implementing our translation system in an object-oriented language: whenever we needed to provide more information to some delegate code generator, we simply assigned a new member to the provider or mandate object. Such extensibility is not however unique to object-oriented languages, and other programming paradigms offer equivalent forms of extension—at the very least the providers, mandate and fragment might have be implemented using a collection type such as a dictionary, in order to achieve the necessary flexibility and extensibility.

Throughout our experimentation, one or other of the providers, mandate or translated fragment proved an appropriate host for any information which needed to be conveyed between code generators. We believe that this is due less to fortune, and more to the inherent elegance of *function-level* abstraction. In those rare cases that information needed to be conveyed which did not pertain either to the function or any argument provider, the mandate served as an effective object of last resort, although in our experience, we never found it necessary to attach ad-hoc data to the mandate. Throughout all of our experiments its use remained within the realm of conveying either general information about the dynamic execution context, or otherwise various desiderata pertaining to the entire translation, such as the convention for numerical representation.

Contextual generation was the most severe test to which we subjected our code generation interface. In this case, specialised code generators were instantiated dynamically during the translation process, and associated as metadata fields on provider objects. This is just one illustration of a powerful pattern, whereby provider objects may be used not just as data, but more generally as delegates for *parametric and context-sensitive* code generation, at the disposal of the code generator to which they are supplied.

Admittedly, our various experiments are at best circumstantial evidence for the sufficiency of the design: applied to other problems our interface for generators might prove inadequate. Certain foreseeable shortcomings are self evident. For example our interfaces are designed for functions taking only a fixed number of positional arguments. To support named or variable arguments a more generic interfaces must be designed. Similarly within this thesis `translate` has only been applied to functions which adhere to structured local control flow. To support structured non-local control flow, such as exceptions, might necessitate a more generic approach.

What is perhaps more immediately concerning however is the *compositionality* of provider and fragment metadata, particularly in foreseeable practical situations. Whilst our interface might prove sufficient in theory, it is another question as to whether is remains feasible for long-term practical use. Whilst the providers supplied to code generators, and the fragments they return, may be extended with arbitrary metadata and methods, how should these data and methods be *organised* such that they do not interfere—and further that their intended influence upon the translation process *composes*? In trivial cases, in which the various metadata each influence the generation of disjoint fragments, compositionality can be achieved simply by establishing a naming convention. So long as different attribute names are assigned for different purposes, then reads and writes to those attributes even on the same object can compose without interference. The more troublesome case surely arises when several extensions to a provider overlap in their intended influence. We have not explored any solutions to this compositionality problem.

6.2.2 Target language extension

Without an intermediate representation language, all optimisations implemented by a late-bound code generator must operate directly on terms in the target language. Typically the target language in question does not natively support all of the intermediate forms required, or at least desired, by the optimiser. For example, when translating procedures to pipelined circuits in Chapter 4, although our synthesisable pipeline model did not support proxy invocation nodes, it was necessary to extend our target language to include non-synthesisable nodes for these invocations, amongst other things, in order to represent intermediate, unoptimised pipeline fragments during the translation process. All such non-synthesisable nodes, which exist only in the extended language, are refined out of the eventual pipeline.

In many cases it is straightforward to extend the target language in this manner, however extending a language in general is a non-trivial problem. In theory the interaction of the extension with all aspects of the unextended language must be considered, and their semantics defined. This imposition of late-bound code generation might, in the worst case, prevent certain languages from being targeted feasibly at all. Since this issue arises from the fundamental definition of the approach, it can only be side-stepped, and cannot be solved. The corresponding advantage of this defining feature, is that code generators have full control over the structure of the eventual translation. Side-stepping the issue might be accomplished by targeting an intermediate language, for which translators to the eventual target independently exist.

6.2.3 Source language breadth

Since the code generators of ReiPy's metacircular interpreter operate directly on terms expressed in the source language—which therefore could include any or all of the features of the source language—they are more complicated and numerous than if a restricted version of the language were used. Steele and Sussman have catalogued a number of transformations for simplifying language terms into restricted forms [SJS76], and which could be applied by ReiPy's preprocessor to simplify the code generation problem. However, the corresponding benefit of operating on full source language terms, is that code generator authors can predict the structure of input terms given only the literal program source.

6.2.4 Generator applicability

In late-bound code generation, the translation of any function's invocation can in principle be overridden, by associating an explicit code generator with that function instance. In practice, the applicability of that overriding code generator is highly sensitive to the translation context—whether or not the overriding code generator is applied to the translation depends on whether or not the callee of the invocation can be unambiguously resolved during the translation. If the callee is not resolved at translation time, a less-specific code generator will be applied, rather than the overriding code generator. This issue might lead to unexpected or undesirable results, and acts as a source of performance non-determinism.

This is an inherent issue in our approach, and is due to the incompleteness of information available (even in theory) to translators posed with the kinds of translation problem we face. The only conceivable general solution is to "weave" some degree of residual translation with the execution of the initial translation result, which could be prohibitive from a performance perspective, or alternatively emit guards into the translation which raise an error if a more-specific code generator is subsequently found to be applicable.

6.3 Comparison with Lancet

Lancet, a framework based on LMS for “surgical-precision” JIT compilation [RSB$^+$14] is closely related to our approach of late-bound code generation. Whilst LMS operates directly on multi-stage programs, Lancet encapsulates the multi-level language front-end of LMS behind a staged metacircular interpreter for Java virtual machine (JVM) bytecode. Combined with the ability to reify bytecode at runtime, it provides a mechanism for explicitly optimising, translating, and offloading computation to heterogeneous hardware, including to GPU accelerators. Furthermore, it features a “JIT macro” system, which allows the compilation of individual methods to be arbitrarily, and separately defined.

One immediate difference between Lancet and our approach can be observed within the rôle of the metacircular interpreter. Without a metacircular interpreter, Lancet provides no mechanism for translating the behaviour of methods—or at least for translating the behaviour of methods not written in the multi-stage language of LMS. In our approach, as illustrated in Chapter 4, code generators are associated directly with closures, and may be used without indirection via an interpreter. Whilst late-bound code generation systems might make use of a metacircular interpreter, as we explored in Chapter 5, to reduce the number of code generators required, they are not a fundamental component. This difference manifests stylistically, when considering how code generators are implemented: code generators for late-bound code generation typically involve several recursive applications of the `translate` operator, whereas macro invocation and the composition of compiled fragments is always managed implicitly by the Lancet framework. Consequently, it is not immediately clear how a Lancet macro could be defined which incorporates the compiled form of a different method, unless it is directly an argument to that macro.

Another, closely related, difference has to do with the compiler interface. Whereas in late-bound code generation `translate` is a globally available operator, which generates and *returns* translated code fragments, in Lancet translations are only immediately available within JIT macros. Consequently late-bound code generation encourages arbitrary generation and composition of code fragments, in the *style* of multi-stage programming languages, yet without adopting their multi-level language semantics. However, in contrast Lancet encourages encapsulation of the translated code within the macro, whilst still providing it an option to distribute or serialise the translation on a case-by-case basis.

Despite these fundamental differences, Lancet shares many similarities with our approach. JIT macros are able to call back into the running program that they are translating, enabling them to take advantage of runtime state, and to perform arbitrary computation including compile-time evaluation of translated terms. Similarly to our code generators, JIT macros are defined separately to the function for which they override the compilation outcome, and thus do not interfere in any way with the definition of the that function. The arguments to a Lancet macro are LMS `Rep` typed values, which are analogous to our providers in that they are representations of translated terms. Unlike our providers, `Rep[T]` values are typically composed using staged versions of the operators which act on `T`, with the ordering of their effects in the translated fragment (let-insertion) managed automatically by LMS—according to compile-time effects emitted by those staged operators. By contrast, in late-bound code generation provider composition and effect ordering is managed explicitly by the code generator, similar in approach to MetaOCaml: the result of the translation is constructed syntactically as a value. Several other points of similarity can be found, for example Lancet's macros are able to conditionally fall-back to the compilation outcome they are overriding, inspect their dynamic scope, and although not explicitly stated in the paper, appear to be selected via dynamic dispatch.

Although Lancet does not have an analogous object to our mandate, Rompf et al. demonstrate how to make use of global state to achieve an equivalent effect—for example using a global variable to control their inlining policy within a delimited dynamic scope. Whilst they do not phrase their description of Lancet in terms of reified scopes, for typical Java objects the fields of an object constitute the closure scope of the object's methods. Since their macros are able to inspect the runtime state of fields, in addition to their associated type declarations, this facility is effectively equivalent to scope reification. They model method-local (invocation) scope as a frame on their bytecode interpreter's stack. Notably, structural operations on this stack, such as pushing and popping frames, are not staged and therefore stack manipulation operations *never* appear within the compiled code. The values stored within frame objects are themselves staged

`Rep` values, which constitutes their solution to the immaterial scope representation problem. Our more elaborate solution based upon contextual generation was necessary only to allow staged scopes to safely escape their initial translation context. Since Lancet does not appear to allow staged scopes to be captured within a macro, their simple solution is apparently sufficient.

An especially sophisticated aspect of Lancet can be found in its use of abstract interpretation: they employ compile-time abstract interpretation of staged values for both type specialisation and constant propagation. In late-bound code generation however, the former is supported by provider metadata, whereas the latter is supported by let-weakening. Their abstract interpretation based approach elegantly unifies the handling of these two concerns, whilst suggesting many more possibilities, for instance supercompilation, as they mention. Lancet's compiler proceeds according to fixed-point iteration, arriving at a translation only after the abstract interpreter stabilises on a least upper-bound over the domain of abstract values. This approach naturally accommodates both non-local control flow, and also a staged form of delimited continuations, neither of which we have explored within late-bound code generation.

Rompf et al. demonstrate a number of diverse applications of Lancet, including both accelerating applications running inside the JVM, and offloading computation to various heterogeneous targets, including GPU accelerators, SQL databases, and web browsers. However they do not explicitly address the introduction of proxies into the offloaded fragment, nor populate anything analogous to our preemption set of Chapter 4. Finally, a number of their applications cover interesting domains to which we have not attempted to apply late-bound generation, namely speculative optimisation, deoptimisation and taint analysis.

6.4 Future work

Although the concrete system for late-bound code generation which we have presented has several limitations, these limitations suggest a number of interesting directions for future work. In this section we outline the most promising potential extensions of our current system, and of the approach more generally.

6.4.1 Non-local control flow

In none of our applications of late-bound code generation did we attempt to handle non-local control flow—even the ability to return early from a function was not explored or supported. It would be imperative for any future work on late-bound code generation to robustly handle the full domain of Python programs, including all forms of non-local control flow, in particular exceptions, coroutines and generators.

We believe that by rewriting ReiPy's metacircular interpreter in continuation-passing style, whilst leaving the interpreted program and code generators in direct style, would be sufficient to handle all non-local control flow operators that currently exist within Python. Some extremely brief and speculative work in this vein has already shown promising results. It is worth noting that when applying `translate` to an operator of a continuation-passing interpreter, the argument provider for the continuation can be supplied symbolically—as an unbound name—to be linked only subsequently in the translation process.

Once non-local control flow is supported in ReiPy we believe that, in combination with let-weakening, it would be capable of forming non-trivial translations for a substantial fraction of valid Python programs. Let-weakening is equally important, since the callee function of an invocation is otherwise frequently unresolvable during translation. This hypothesis could be tested, by applying ReiPy to a diverse sample of valid programs, and inspecting the quality of the translations generated.

Currently ReiPy never includes translations of functions for which it does not discover a potential invocation. In certain situations a translation will be generated with proxy objects corresponding to terms of the metacircular interpreter, which results in the translated fragment calling back into the interpreter of the originating process. It would be interesting to further reduce the extent to which this phenomenon occurs, by including an implementation of the Python interpreter in the translated fragment, along with other immutable information and translations of all closures present in the originating process.

So far we have not encountered any evidence to suggest that the interface for code generators, or the `translate` operator for that matter, need to be recast to continuation-passing style, in general. It is, however, conceivable that certain potential abstractions, particularly those used by code generators in the formation of their result, might necessitate this generalisation. Further experimental work would be required to determine if this issue arises in practice.

6.4.2 Imperative propagation of information

During execution of an imperative program, information typically propagates from one statement to another via side-effects in the program's mutable state—for example variable assignments or object field mutations. We introduced let-weakening in order to make certain properties of this information flow more apparent to our translation system, specifically eliminating side-effects due to variable assignment where possible.

It would be desirable to investigate other potential mechanisms for propagating this (partial) information during translation. There is an implicit design objective in late-bound code generation systems to keep the input and output representations of the code generators as close as possible to the structure of the source program and target fragment. This is because an application developer wishing to override the translation of some aspect of their application will have a concrete conception of the source and target forms, but will not be cognisant of arbitrary intermediate representations created by ReiPy's internal implementation. Let-weakening disturbs the structure of the program which is acted upon by the metacircular interpreter, and so replacing it with another information propagation solution, that does not disturb this structure, would be preferable. An approach which makes use of, and contributes to, provider metadata might be possible.

6.4.3 Unified intermediate representation

Although, as we argued in the previous subsection, it is desirable for code generators to take and return representations which mirror source and target terms, it could be worth investigating a unified intermediate representation for late-bound code generation, that serves as a common intermediate target language for translations. This would afford an opportunity to design a representation which is well suited to certain kinds of common optimisation, such as inlining and constant folding, and which might also provide a richer basis for explicitly encoding control-flow, data-flow and type information.

In most situations an application developer cares only about a minority of aspects pertaining to the translation: some general concerns include behavioural correctness, average performance and code size. However, they also typically care about a limited number of application specific concerns, for example the latency of certain program transitions. To retain pragmatic control within these situations, it would be necessary to include in the intermediate representation language an ability to commit fragments to a specific eventual target term: akin to the `asm` statement in the C programming language.

6.4.4 Metadata compositionality

Our translation system makes use of metadata properties on both the mandate and the provider objects. The experiments we undertook in this thesis were sufficiently simple that the issue of metadata compositionality did not occur in practice. However, it is clear that in a more complicated application, which makes use of a diverse variety of separate and interacting metadata properties, the issue of how such properties are organised and composed would become manifest.

The space of potential abstractions for organising and composing mandate and provider metadata is clearly quite vast. It would be interesting to apply ReiPy to a problem which is more demanding of its metadata attributes. A problem of this kind might, for example, include functions and data structures which independently contribute requirements for bit-representations, memory layouts, and inlining and parallelism objectives. This would help to inform which metadata abstractions are worthy of further investigation.

6.4.5 Generalising `translate`

In order to accommodate the translation of functions which accept anything other than a fixed number of positional arguments, the interface for `translate`, and also of its code generators must be generalised. It would seem appropriate to merge the function provider and argument providers all into a single provider, which provides a single object comprising all of this information—this object is naturally extensible (since it is an object) and could for example be extended to include variable and optional named arguments. The principle of contextual generation could then be leveraged, to allow code generators to *generate* the function provider, a particular argument provider, or argument list: thus retaining at least the same capabilities as our current interface. This comes at the cost of greater abstraction and complexity within the translation process.

We suggest a thorough investigation into the potential generalisations of the dispatch policy for `translate`, possibly also including customisable policies, which could be controlled via the mandate. Our experiments to date have suggested at least four categories of cite to which code generators might be associated: (i) a provider (as in contextual generation); (ii) a function or method instance; (iii) a function or method definition / class definition; and (iii) the target language. Broadly speaking, these categories are listed in order from most specific to least specific, although such an ordering is somewhat arbitrary. The dispatch policy of `translate` used within this thesis simply delegates translation to the most specific generator according to this list, falling back to less specific generators when required. It would be interesting to investigate the full range of potential association cites, and also the various kinds of policy which can be defined over this set.

Furthermore, in our current conception, the code generator for a function is responsible for generating a complete implementation of the function's behaviour in the target language: it typically fulfills this responsibility by delegating the translation of certain sub-terms to other code generators, via recursive application of `translate`. Different kinds of code generator might also be conceived of—which could even coexist alongside our original conception—that contribute to the translation, but are not wholly responsible for it, or that emit some representation for a range of acceptable translations, which are subsequently selected from.

6.4.6 Structurally separating convention from language

Although we have regularly referred to the first parameter of `translate` as the target *language*, this conflates a pair of distinct concepts. The target of the translation defines the target language, but it also defines a set of conventions applied within the target language, which are not strictly properties of the language itself. Examples of these conventions include, how certain classes of object are represented, and how function invocations are performed within the target fragment. The target language itself usually permits a wide array of different such possibilities, but for the fragments generated by separate code generators to compose properly, they must all adhere to a common set of these conventions.

This issue is directly analogous to the various C ABIs that have been defined, independently from the C language specification. It seems prudent that future work on late-bound code generation structurally separates the target language from these conventions. This is a rather technical point, since the two separate instances of target language and target conventions might still be composed into one target object, which is retained as the first parameter to `translate`.

6.4.7 Static compilation

Whilst late-bound code generation is primarily intended as a technique for runtime code generation, it would be worth comparing the output of a late-bound code generation system with that of a traditional compiler, if a compiler for the source language is available. Such an experiment could be constructed by simply writing a program which binds various functions or class definitions to names within the global scope, including some “main” function, and then applying `translate` to that main function. The resulting translation should be comparable with that of the static compiler, and any deficiencies noted would inform where to expend further effort on improvements.

6.4.8 Use-centric abstractions

Even though ReiPy might be improved to the point that it can generate acceptable translations for any Python function, and to an array of desirable target languages, this does not necessarily mean that it has any direct utility. It is conceivable that for every problem to which it might be applied, there is also substantial effort required to integrate the interfaces presented by ReiPy to those abstractions in terms of which the problem is conventionally solved.

For example, to apply ReiPy to the challenge of offloading numerical computations to a GPU would entail more that generating GPU kernel code. The following problems also need to be solved: (i) installing the kernel in the GPU; (ii) marshalling input data into the format expected by the kernel; (iii) allocating the necessary memory space within the CPU and GPU address space; (iv) feeding the data it into the GPU; (v) unmarshalling the results; and potentially (vi) encapsulating all of these operations within a functional interface. Furthermore, if the developer of such an application were expected to write or customise a number of code generators, it might be important to provide suitable abstractions to assist them, and to automatically enforce any constraints which are generally relied upon for the translation to be valid.

Collating a set of compelling, diverse use-cases for late-bound code generation would greatly inform any effort at designing its requisite use-centric abstractions, and implementing these additional features and libraries. Such an effort might also entail standardising interfaces at which components of the system interact, for instance defining a standard protocol, comprising a data and control language, for use by the channel connecting the host and target processes. One especially interesting token-based approach to formalising the protocol for such a channel has been suggested by Fredriksson and Ghica [FG13]

6.4.9 Alternative circuit models

When examining the statically scheduled rasterisation circuit of Chapter 4 we noted that it included a large number of deep queues, used to synchronise data values in the absence of any control signalling, other than a global clock. We also observe that the restriction of a static schedule in the translation imposes a severe restriction on the translated program: it must at least be free of any unbounded recursion or iteration. Extending the circuit model to include a modicum of additional control signalling might simultaneously reduce the inefficiency due to these deep queues, and also admit a wider translation domain.

We suspect that some variation on a token-based circuit model might be well suited to this end, and would certainly be worth investigating. Ghica et al. [Ghi11, GSS11] describe a circuit model which features a single-bit control token: this token is passed around the circuit and corresponds to a distribution of the von Neumann model program counter. Their model supports recursion via the introduction of localised stack memories, although pipeline parallelism is not supported. Other variations on this token-based model includes a "ready" bit, allowing circuit components to reject control tokens during high-latency operations, for example during shared memory accesses [CRKE15]. Townsend et al. [TKE17] employ this extended control interface within their circuit model, which supports pipelining in addition to recursion. In contrast to Ghica's approach, their model however allocates control information on a global heap. Combining the local stack memories of the former model, with the pipelining support from the latter, would be of particular interest.

With an improved circuit model at hand, it would also be worth considering which coarse-grained reconfigurable architectures (CGRAs) most efficiently support that model. It is possible that certain existing CGRA architectures, for example various accelerators designed for machine learning applications, would outperform an FPGA in this regard. Another possibility is that the restrictions of the circuit model inform a novel CGRA architecture, which might serve as a basis for better accelerating programmatically defined, pipelined computations.

6.4.10 Broader directions

One broad direction for future research would be to explore wholly different approaches to syntactification. Late-bound code generation, as defined in this thesis, performs syntactification according to a deterministic, recursively ordered and dynamically dispatched code generation model. It would be interesting to explore fundamentally distinct techniques, for instance performing elaboration by the non-deterministic application of pattern matching rewrite rules.

Finally, our experience with late-bound code generation suggests that it would be well suited to the optimisation and translation of F-expression based languages, such as for example Shutt's programming language Kernel [Shu10].

Appendix A

Glossary

Closure—A combination of a syntactical function definition, and a runtime environment (scope) over which names free within the definition are closed.

Function definition—A fragment of program syntax which defines a function.

Function instance—see **Closure**.

Hygiene—Ensuring that name bindings never resolve to an *unintended* value: splicing a code fragment which assigns to a temporary variable via a name shared with the surrounding code is an example of hygiene violation, since subsequent surrounding uses of that name would be rebound to the unintended temporary value. Hygiene may be maintained either manually by the programmer, or automatically within the language system.

Late-bound code generation—A technique for performing syntactification, based upon dynamically dispatched mutual recursion between fragmentary code generators.

Let insertion—A discipline for maintaining the order of side-effects, and reducing re-computation in generated code, by inserting "let" bindings, in order to bring the code into administrative normal form (ANF). For a detailed introduction to this technique see [Rom16].

Let-weakening—A novel transformation for weakening mutable to immutable named let bindings, described in Section 5.3.1.

Pragmatic control—The control of aspects of a program's execution which do not bear on the correctness of the values which it produces: for instance controlling which hardware unit is used to execute a particular section of the program, which machine instructions are used to implement a particular operation, or how instructions are scheduled in time.

Pragmatic requirement—A requirement for a certain section of a program to execute within a particular hardware unit, for an operation to be implemented using certain machine instructions, or a restriction on how machine instructions are scheduled in time.

Pragmatic self-control—A program exerting pragmatic control over its *own* execution.

Pragmatic statement—A statement in a program which affects pragmatic control over another program, or the same program; the latter being a case of *pragmatic self-control*.

Reification—To make some aspect of a program's behaviour or state accessible to the program itself, typically as a runtime value. For example a section of a program's behaviour might be reified as list of machine instructions, or the state of a particular scope might be reified as a dictionary from strings to values.

Self-issuance—When a program explicitly issues instructions to hardware for performing its own computation, rather than implicitly relying on the language system to issue hardware instructions on its behalf.

Syntactification—An approach to translation whereby translated terms are progressively refined from simpler less elaborate forms, to complex more elaborate forms. It operates via a process of iterative reduction over syntactic closures in the target language: reduction steps discover binding and behavioural information implicit within the syntactic closure, and representing that syntactically, typically inline within the translation, transforming information from scope to syntax.

Bibliography

[ABC$^+$16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[AI82] Atari Incorporated. C012296 ANTIC (NTSC) Datasheet Revision D. https://web.archive.org/web/20040909041445/http://www.retromicro.com/files /atari/8bit/antic.pdf, 1982.

[AMP$^+$19] Akshay Agrawal, Akshay Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning. In *Proceedings of Machine Learning and Systems*, volume 1, pages 178–189, 2019.

[App98] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, April 1998.

[AR17] Nada Amin and Tiark Rompf. LMS-Verify: Abstraction without regret for verified systems programming. In *Proceedings of the 44th ACM SIGPLAN Sympo-*

sium on Principles of Programming Languages, POPL 2017, page 859–873, New York, NY, USA, 2017. Association for Computing Machinery.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*, page 1–11, New York, NY, USA, 1988. Association for Computing Machinery.

[Baw99] Alan Bawden. Quasiquotation in Lisp. In *PEPM*, 1999.

[Ber14] David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[BFD08] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs. In Laurie Hendren, editor, *Compiler Construction*, pages 178–192, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Boh07] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[BPRS18] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

[BR88] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, page 86–95, New York, NY, USA, 1988. Association for Computing Machinery.

[BSL$^+$11] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 89–100, 2011.

[BTU17] Martin Berger, Laurence Tratt, and Christian Urban. Modelling Homogeneous Generative Meta-Programming. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:23, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[CA11] Davor Capalija and Tarek S. Abdelrahman. Towards synthesis-free JIT compilation to commodity FPGAs. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 202–205, 2011.

[CBC$^+$05] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 272–283, 2005.

[CGMT09] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[CH10] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[CHK01] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[CR05] William R. Cook and Siddhartha Rai. Safe query objects: Statically typed objects as remotely executable queries. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 97–106, New York, NY, USA, 2005. Association for Computing Machinery.

[CRKE15] Bingyi Cao, Kenneth A. Ross, Martha A. Kim, and Stephen A. Edwards. Implementing latency-insensitive dataflow blocks. In *2015 ACM/IEEE International*

Conference on Formal Methods and Models for Codesign (MEMOCODE), pages 179–187, 2015.

[CTHL03] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering (GPCE '03)*, pages 57–76. Springer, Berlin Heidelberg, 2003.

[CW20] William D. Clinger and Mitchell Wand. Hygienic macro technology. *History of Programming Languages (HOPL)*, June 2020.

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A hybrid multicore parallel programming environment. *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-1)*, 2007.

[DM98] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[DWS+88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, page 21–30, New York, NY, USA, 1988. Association for Computing Machinery.

[ETD+18] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. Flare: Optimizing Apache Spark with native compilation for scale-up architectures and medium-size data. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 799–815, Carlsbad, CA, October 2018. USENIX Association.

[FG13] Olle Fredriksson and Dan R. Ghica. Seamless distributed computing from the geometry of interaction. In *Trustworthy Global Computing*, pages 34–48, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Fou19] Python Software Foundation. Bug Tracker Issue 20443: `__code__.co_filename` should always be an absolute path. `https://bugs.python.org/issue20443`, 2019.

[Fou21] Python Software Foundation. Python Language Services: Abstract Syntax Trees (Python 3.9). https://docs.python.org/3.9/library/ast.html, 2021.

[FR82] Donald Fussell and Bharat Deep Rathi. A VLSI-oriented architecture for real-time raster display of shaded polygons. Graphics Interface, 1982.

[Fut99] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[FvDFH96] James D Foley, Andries van Dam, Steven K Feiner, and John F Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 1996.

[GGSS89] N. Gharachorloo, S. Gupta, R. F. Sproull, and I. E. Sutherland. A characterization of ten rasterization techniques. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '89)*, page 355–368, New York, NY, USA, 1989. Association for Computing Machinery.

[Ghi11] Dan R. Ghica. Function interface models for hardware compilation. In *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2011)*, pages 131–142, 2011.

[GLN+14] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014.

[GM10] Srinivas Guruzu and Gary Mak. HQL and JPA Query Language. *Hibernate Recipes: A Problem-Solution Approach*, pages 155–166, 2010.

[GRR$^+$14] Ivan Grasso, Petar Radojkovic, Nikola Rajovic, Isaac Gelado, and Alex Ramirez. Energy efficient HPC on embedded SoCs: Optimization techniques for Mali GPU. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 123–132, 2014.

[GSS11] Dan R. Ghica, Alex Smith, and Satnam Singh. Geometry of Synthesis IV: Compiling affine recursion into static hardware. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 221–233, New York, NY, USA, 2011. Association for Computing Machinery.

[GW14] Jeremy Gibbons and Nicolas Wu. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *International Conference on Functional Programming*, ICFP '14, page 339–347, New York, NY, USA, 2014. Association for Computing Machinery.

[Hor14] Mark Horowitz. Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[HVO$^+$06] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, volume 1, pages 13–15. IEEE, 2006.

[HWZ14] David Herman, Luke Wagner, and Alon Zakai. asm.js Working Draft. http://asmjs.org/spec/latest/, 2014.

[Jon10] Simon Peyton Jones. New directions for Template Haskell. https://www.haskell.org/ghc/blog/20101018-Template%20Haskell%20Proposal.html, 2010.

[Kel95] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. *SIGPLAN Notices*, 30(3):13–22, March 1995.

[KKS08] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Closing the stage: From staged code to typed closures. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '08, page 147–157, New York, NY, USA, 2008. Association for Computing Machinery.

[KKS09] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: Staging with delimited control. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, page 111–120, New York, NY, USA, 2009. Association for Computing Machinery.

[KST04] Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, page 249–258, New York, NY, USA, 2004. Association for Computing Machinery.

[LI22] Google LLC and Skia Inc. Skia documentation. https://skia.org/docs, 2022.

[Loc79] Bart Locanthi. Object oriented raster displays. Proceedings of the Caltech Conference On Very Large Scale Integration. California Institute of Technology, Pasadena, CA, 1979.

[LSV04] Roman Lysecky, Greg Stitt, and Frank Vahid. Warp processors. *ACM Transactions on Design Automation of Electronic Systems*, 11(3):659–681, June 2004.

[LtAOSP22] Google LLC and the Android Open Source Project. Android API reference. https://developer.android.com/reference, 2022.

[MAA16] Sen Ma, Zeyad Aklah, and David Andrews. Just in time assembly of accelerators. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 173–178, New York, NY, USA, 2016. Association for Computing Machinery.

[MBEP13] Arian Maghazeh, Unmesh D. Bordoloi, Petru Eles, and Zebo Peng. General purpose computing on low-power embedded GPUs: Has it come of age? In

2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pages 1–10, 2013.

[MDW$^+$19] Dan Moldovan, James Decker, Fei Wang, Andrew Johnson, Brian Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. AutoGraph: Imperative-style coding with graph-based performance. In *Proceedings of Machine Learning and Systems*, volume 1, pages 389–405, 2019.

[Mos70] Joel Moses. The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem. *SIGSAM Bulletin*, page 13–27, July 1970.

[MS09] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[MTIMPP19] Microchip Technology Incorporated (Microsemi Product Portfolio). Datasheet: DS0141, 2019.

[MTIMPP21] Microchip Technology Incorporated (Microsemi Product Portfolio). Documentation / Power Estimation / Polarfire and PolarFire SoC Power Estimator. https://www.microsemi.com/product-directory/fpgas/3854-polarfire-fpgas, 2021.

[MTIMPP22] Microchip Technology Incorporated (Microsemi Product Portfolio). Documentation / Low Power Leadership. https://www.microsemi.com/product-directory/fpga-soc/1743-low-power, 2022.

[MV14] Sparsh Mittal and Jeffrey S. Vetter. A survey of methods for analyzing and improving GPU energy efficiency. *ACM Comput. Surv.*, 47(2), aug 2014.

[NCPC11] NVIDIA, Cray, PGI, and CAPS. ‘OpenACC’ Programming Standard for Parallel Computing. In *Proceedings of the ACM International Conference on Supercomputing*, 2011.

[ORS$^+$13] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the systematic construction of generators for performance libraries. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, page 125–134, New York, NY, USA, 2013. Association for Computing Machinery.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 199–208, New York, NY, USA, 1988. Association for Computing Machinery.

[PJTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. *ACM SIGPLAN Haskell Workshop*, 2001.

[Ris18] Marko Ristin. Answer to StackOverflow question: "How can I get the source code of a Python function?". `https://stackoverflow.com/a/52333691`, 2018.

[RMJ$^+$21] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. AI accelerator survey and trends. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2021.

[RO10] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*, page 127–136, New York, NY, USA, 2010. Association for Computing Machinery.

[Rom12] Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming.* PhD thesis, EPFL, 2012.

[Rom16] Tiark Rompf. *The essence of multi-stage evaluation in LMS*, pages 318–335. Springer International Publishing, Cham, 2016.

[Roo02] Robin Roos. *Java Data Objects*. Pearson Education, 2002.

[RSB+14] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. Surgical precision JIT compilers. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 41–52, New York, NY, USA, 2014. Association for Computing Machinery.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery.

[SC88] Bengt-Olaf Schneider and Ute Claussen. Proof: An architecture for rendering in object space. In *Proceedings of the Third Eurographics Conference on Advances in Computer Graphics Hardware*, EGGH'88, page 121–140, Goslar, DEU, 1988. Eurographics Association.

[Sch88] Bengt-Olaf Schneider. A processor for an object-oriented rendering system. *Computer Graphics Forum*, 7(4):301–310, 1988.

[SGB+13] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a high performance DSL implementation from a declarative specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, page 145–154, New York, NY, USA, 2013. Association for Computing Machinery.

[Shu10] John N. Shutt. *Fexprs as the basis of Lisp function application; or, **$vau**: the ultimate abstraction*. PhD thesis, Worcester Polytechnic Institute, 2010.

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, page 1–16, New York, NY, USA, 2002. Association for Computing Machinery.

[SJS76] Guy Lewis Steele Jr. and Gerald Jay Sussman. Lambda: The Ultimate Imperative. Massachusetts Institute of Technology AIM-353, 1976.

[SKK18] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. Building efficient query engines in a high-level language. *ACM Transactions on Database Systems*, 43(1), April 2018.

[SLB$^+$11] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning. In *ICML*, pages 609–616, 2011.

[TKE17] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. From functional programs to pipelined dataflow circuits. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 76–86, New York, NY, USA, 2017. Association for Computing Machinery.

[TN03] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 26–37, New York, NY, USA, 2003. Association for Computing Machinery.

[Tra05] Laurence Tratt. Compile-time meta-programming in a dynamically typed OO language. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS '05, page 49–63, New York, NY, USA, 2005. Association for Computing Machinery.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1):211–242, 2000. PEPM'97.

[Vas08] Yuli Vasiliev. Using Java Persistence Query Language (JPQL). *Beginning Database-Driven Application Development in Java™ EE: Using GlassFish™*, pages 283–302, 2008.

[Ver21] Veripool. Verilator (Verilog simulator). https://www.veripool.org/verilator, 2021.

[Vis99] Eelco Visser. Strategic pattern matching. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, page 30–44, Berlin, Heidelberg, 1999. Springer-Verlag.

[Wei81] Richard Weinberg. Parallel processing image synthesis and anti-aliasing. In *Proceedings of the 8th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '81, page 55–62, New York, NY, USA, 1981. Association for Computing Machinery.

[WGSD07] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. *ACM Transactions on Software Engineering and Methodology*, 16(4), September 2007.

[WRI$^+$10] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 400–411, New York, NY, USA, 2010. Association for Computing Machinery.

[Xil21] Xilinx. Vitis Unified Software Development Platform 2021.1 Documentation. https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/hls_pragmas.html, 2021.