A Statically Allocated Parallel Functional Language

Alan Mycroft[1,2] and Richard Sharp[2]

[1] Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK
am@cl.cam.ac.uk
[2] AT&T Laboratories Cambridge
24a Trumpington Street, Cambridge CB2 1QA, UK
rws@uk.research.att.com

Abstract. We describe SAFL, a call-by-value first-order functional language which is syntactically restricted so that storage may be statically allocated to fixed locations. Evaluation of independent sub-expressions happens in parallel—we use locking techniques to protect shared-use function definitions (i.e. to prevent unrestricted parallel accesses to their storage locations for argument and return values). SAFL programs have a well defined notion of total (program and data) size which we refer to as 'area'; similarly we can talk about execution 'time'. Fold/unfold transformations on SAFL provide mappings between different points on the area-time spectrum. The space of functions expressible in SAFL is incomparable with the space of primitive recursive functions, in particular interpreters are expressible. The motivation behind SAFL is hardware description and synthesis—we have built an optimising compiler for translating SAFL to silicon.

1 Introduction

This paper addresses the idea of a functional language, SAFL, which

- can be statically allocated—all variables are allocated to fixed storage locations at compile time—there is no stack or heap; and
- has independent sub-expressions evaluated concurrently.

While this concept might seem rather odd in terms of the capabilities of modern processor instruction sets, our view is that it neatly abstracts the primitives available to a hardware designer. Our desire for static allocation is motivated by the observation that dynamically-allocated storage does not map well onto silicon: an addressable global store leads to a von Neumann bottleneck which inhibits the natural parallelism of a circuit. SAFL has a call-by-value semantics since strict evaluation naturally facilitates parallel execution which is well suited to hardware implementation.

U. Montanari et al. (Eds.): ICALP 2000, LNCS 1853, pp. 37–48, 2000.

To emphasise the hardware connection we define the *area* of a SAFL program to be the total space required for its execution. Due to static allocation we see that *area* is $O(\textit{length of program})$; similarly we can talk about execution *time*. Fold/unfold transformations [1] at the SAFL level correspond directly to area-time tradeoffs at the hardware level.

In this paper we are concerned with the properties of the SAFL language itself rather than the details of its translation to hardware. In the light of this and for the sake of clarity, we present an implementation of SAFL by means of a translation to an abstract machine code which we claim mirrors the primitives available in hardware. The design of an optimising compiler which translates SAFL into hardware is presented in a companion paper [11]. A more practical use of SAFL for hardware/software co-design is given in [9].

The body of this paper is structured as follows. Section 2 describes the SAFL language and Section 3 describes an implementation on a parallel abstract machine. In Sections 4 and 5 we argue that SAFL is well suited for hardware description and synthesis. Section 6 shows how fold/unfold transformations can represent SAFL area-time tradeoffs. Finally, Sections 7 and 8 discuss more theoretical issues: how SAFL relates to Primitive Recursive functions and problems concerning higher-order extensions. Section 9 concludes and outlines some future directions.

Comparison with Other Work

The motivation for static allocation is not new. Gomard and Sestoft [2] describe *globalization* which detects when stack or heap allocation of function parameters can be implemented more efficiently with global variables. However, whereas globalization is an optimisation which may in some circumstances improve performance, in our work static allocation is a fundamental property of SAFL enforced by the syntactic restrictions described in Section 2.

Previous work on compiling declarative specifications to hardware has centred on how functional languages themselves can be used as tools to aid the design of circuits. Sheeran's muFP [12] and Lava [13] systems use functional programming techniques (such as higher order functions) to express concisely the repeating structures that often appear in hardware circuits. In this framework, using different interpretations of primitive functions corresponds to various operations including behavioural simulation and netlist generation. Our approach takes SAFL constructs (rather than gates) as primitive. Although this restricts the class of circuits we can describe to those which satisfy certain high-level properties, it permits high-level analysis and optimisation yielding efficient hardware. We believe our association of function definitions with hardware resources (see Section 4) to be novel.

Various authors have described silicon compilers (e.g. for C [4] and Occam [10]). Although rather beyond the scope of this paper, we argue that the flexibility of functional languages provides much more scope for analysis and optimisation.