Generating Heap-bounded Programs in a Functional Setting*

Walid Taha[1], Stephan Ellner[1], and Hongwei Xi[2]

[1] Rice University, Houston, TX, USA
{taha,besan}@cs.rice.edu
[2] Boston University, Boston, MA, USA
hwxi@cs.bu.edu

Abstract. High-level programming languages offer significant expressivity but provide little or no guarantees about resource utilization. Resource-bounded languages provide strong guarantees about the runtime behavior of programs but often lack mechanisms that allow programmers to write more structured, modular, and reusable programs. To overcome this basic tension in language design, this paper advocates taking into account the natural distinction between the development platform and the deployment platform for resource-sensitive software.
To illustrate this approach, we develop the meta-theory for GeHB, a two-level language in which first stage computations can involve arbitrary resource consumption, but the second stage can only involve functional programs that do not require new heap allocations. As an example of a such a second-stage language we use the recently proposed first-order functional language LFPL. LFPL can be compiled directly to malloc-free, imperative C code. We show that all generated programs in GeHB can be transformed into well-typed LFPL programs, thus ensuring that the results established for LFPL are directly applicable to GeHB.

1 Introduction

Designers of embedded and real-time software attend not only to functional specifications, but also to a wider range of concerns, including resource consumption and integration with the physical world. Because of the need for strong a-priori guarantees about the runtime behavior of embedded software, resource-bounded languages (c.f. [19, 8, 9, 28, 29]) generally trade expressivity for guarantees about runtime behavior. Depending on the kind of guarantees required, a resource-bounded language may have to deprive the programmer from useful abstraction mechanisms such as higher-order functions, dynamic data structures, and general recursion. We argue that this trade-off can be avoided if the language itself can express the distinction between computation on the development platform and computation on the deployment platform. Such a language could provide a bridge between traditional software engineering techniques on one side, and the specific demands of the embedded software domain on the other.

* Supported by NSF grants ITR-0113569, CCR-0224244 and CCR-0229480.

LFPL: Bounded Space and Functional In-place Update Recently, Hofmann proposed the resource-bounded programming language LFPL [8, 10], a first-order functional language with constructors and destructors for dynamic data structures. The type system of LFPL ensures that all well-typed programs can be compiled into malloc-free, imperative C code. [8]. Without any special optimizations, the performance of resulting programs is competitive with the performance of programs generated by the OCaml native code compiler.

The essential idea behind LFPL is the use of a linear type system that ensures that references to dynamically allocated structures are not duplicated at runtime. The key mechanism to achieving this is ensuring that certain variables in the source program are used at most once (linearity). Constructors for dynamic structures carry an extra field that can be informally interpreted as a capability. The following is an implementation of insertion sort over lists in LFPL [10]: [3]

```
let rec insert(d,a,l) =
    case l of
      nil -> cons(a, nil) at d
    | cons(b,t) at d' -> if a < b
                         then cons(a, cons(b, t) at d') at d
                         else cons(b, insert(d',a, t)) at d
let rec sort(l) =
  case l of
    nil -> nil
  | cons(a,t) at d -> insert(d, a, sort(t))
```

The main function, `sort`, takes a list `l` and returns a sorted list. Using pattern matching (the `case`-statement), the function checks if the list is empty. If so, the empty list is returned. If the list is non-empty, the constructor `cons` for the list carries three values: the head `a`, the tail `t`, and a capability `d` for the location at which this list node is stored. Note that the capability `d` is passed to `insert`, along with the head of the list and the result of recursively calling `sort` on the tail. The type system ensures that well-typed programs do not duplicate such capabilities. This is achevied by ensuring that a variable like `d` is not used more than once.

It has been shown that any LFPL program can be compiled to a C program that requires no heap allocation, and therefore no dynamic heap management [8, 10]. Instead of allocating new space on the heap, constructor applications are implemented by modifying heap locations of previously allocated data structures.

Expressivity The example above points out a common limitation of resource-bounded languages. In particular, we often want to parameterize sorting functions by a comparison operation that allows us to reuse the same function to sort different kinds of data (c.f. [14]). This is especially problematic with larger and more sophisticated sorting algorithms, as it would require duplicating the source code for such functions for every new data structure. In practice, this can also cause code duplication, thus increasing maintenance cost.

[3] We use an OCaml-like concrete syntax for LFPL.

Such parameterization requires higher-order functions, which are usually absent from resource-bounded languages: implementing higher-order functions requires runtime allocation of closures on the heap. Furthermore, higher-order functions are expressive enough to encode dynamically allocated data structures such as lists.

Contributions A key observation behind our work is that even when targeting resource-bounded platforms, earlier computations that lead to the *construction* of such programs are often not run on resource-bounded development platforms. Commercial products such as National Instrument's LabVIEW Real-Time [20] and LabVIEW FPGA [21] already take advantage of such stage distinctions. The question this paper addresses is how to reflect this reality into the design of a language that gives feedback to the programmer as soon as possible as to whether the "final product" is resource-bounded or not.

We demonstrate that ideas from two-level and multi-stage languages can be used to develop natural, two-stage extensions of a resource-bounded language. The focus of this paper is on developing the meta-theory for GeHB, a statically-typed two-stage language that extends LFPL. The resulting language is more expressive and takes advantage of the realistic assumption of having two distinct stages of computation. In the first stage, general higher-order functions can be used. In the second (or "LFPL") stage, no dynamic heap allocation is allowed. Type-checking a GeHB program (which happens before the first stage) statically guarantees that all second-stage computations are heap-bounded. This strong notion of static typing requires that we type check not only first-stage computations, but also templates of second-stage programs. Compared to traditional statically-typed multi-stage languages, the technical challenge lies in ensuring that generated programs satisfy the linearity conditions that Hofmann defines for LFPL. While a direct combination of a multi-stage language and LFPL does not work, we show how this problem can be addressed using the recently proposed notion of closed types [4, 7]. Finally, we show that all generated programs can be transformed into well-typed LFPL programs, thus ensuring that the results established for LFPL are directly applicable to GeHB.

The results presented here are a step toward our long-term goal of developing an expressive statically-typed functional language that can be used in the challenging context of embedded systems [27].

Organization of this Paper Section 2 introduces the basic concepts of general-purpose multi-stage programming, and points out a technical problem in the direct combination of a multi-stage type system with the type system of LFPL. Section 3 formalizes the type system and semantics of GeHB, and presents a type preservation result for first-stage evaluation. The section also shows that second-stage values determine LFPL programs and that results about LFPL apply to GeHB. Section 4 illustrates programming in GeHB and the impact of typing rules from the programmer's perspective. Sections 5 and 6 discuss various aspects of the work and conclude.

2 Multi-Stage Programming

Multi-stage languages [22, 13, 24] provide light-weight, high-level *annotations* that allow the programmer to break down computations into distinct *stages*. This facility supports a natural and algorithmic approach to program generation, where generation occurs in a first stage, and the synthesized program is executed during a second stage. The annotations are a small set of constructs for the construction, combination, and execution of delayed computations. Underlying program generation problems, such as avoiding accidental variable capture and the representation of programs, are completely hidden from the programmer (c.f. [24]). The following is a simple program written in the multi-stage language MetaOCaml [17]:

```
let rec power n x = if n=0 then <1> else <~x * ~(power (n-1) x)>
let power3 = <fun x -> ~(power 3 <x>)>
```

Ignoring the staging annotations (brackets `<`e`>` and escapes `~`e) [4], the above code is a standard definition of a function that computes x^n, which is then used to define the specialized function x^3. Without staging, the last step just produces a closure that invokes the power function every time it gets a value for x. The effect of staging is best understood by starting at the end of the example. Whereas a term `fun x ->` e `x` is a value, an annotated term `<fun x -> ~(`e `<x>)>` is not. Brackets indicate that we are constructing a future stage computation, and an escape indicates that we must perform an immediate computation *while* building the bracketed computation. The application e `<x>` has to be performed even though `x` is still an uninstantiated symbol. In the `power` example, `power 3 <x>` is performed immediately, once and for all, and not repeated every time we have a new value for `x`. In the body of the definition of the function `power`, the recursive application of `power` is escaped to ensure its immediate execution in the first stage. Evaluating the definition of `power3` produces `<fun x -> x*x*x*1>`.

General-purpose multi-stage languages provide strong safety guarantees. For example, a program generator written in such a language is not only type-safe in the traditional sense, but the type system also guarantees that *any generated program will be type safe*. Traditionally, multi-stage languages were used for quantitative benefits (speed-up). In this work, the benefit is more qualitative, in that it allows programmers to write programs that previously could not be expressed in a resource-bounded language.

A Naive Approach to Generating LFPL Programs Combining the type systems for multi-stage languages and LFPL is not trivial. In particular, it cannot be achieved by simply treating level 0 variables as non-linear variables (that can be used multiple times). The level of a term e is 0 if it is executed in the first stage, or 1 if it is executed in the second. A term is at level 0 by default, unless it occurs (unescaped) inside brackets. For example, consider the expression `<fun x -> ~(power 3 <x>)>`. The use of `power` is surrounded by

[4] In the implementation of MetaOCaml, dots are used around brackets and before escapes to disambiguate the syntax.

one set of brackets and one escape, therefore `power` is used at level 0. The use of `x` is surrounded by two sets of brackets and one escape, and therefore occurs at level 1. Similarly, the binding of `x` occurs at level 1. Treating level 0 variables as non-linear and level 1 variables as linear is problematic for two reasons:

1. *Some level 0 variables must be treated linearly:* Variables bound at level 0 can refer to computations that contain free level 0 variables. A simple example is the following:

   ```
   <fun x -> ~((fun y -> <(~y,~y)>) <x>)>
   ```

   In this term, `x` is bound at level 1, while `y` is bound at level 0. Note that `x` is used only once in the body of the term, and is therefore linear. However, if we evaluate the term (performing the first stage computation) we get the term

   ```
   <fun x -> (x,x)>
   ```

   where `x` is no longer used linearly. This occurs because `y` was not used linearly in the original term. Thus, one must also pay attention to linearity even for level 0 variables such as `y`.
2. *Not all level 1 variables need to be linear:* Hofmann's type system is parameterized by a signature of function names. Such functions can be used many times in an LFPL program. A natural way to integrate such function names into our formulation is to treat them as non-linear variables. [5]

Ignoring the first point leads to an unsound static type system: it fails to ensure that generated programs require no heap allocation in the second stage. Ignoring the second leads to an overly restrictive type system.

3 GeHB: Generating Heap Bounded Programs

The syntax of the subset of GeHB that we study is defined as follows:

$$e ::= i \mid x \mid \lambda x.e \mid e(e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid (e,e) \mid \pi_1\ e \mid \pi_2\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{let\ rec}\ f(x) = e\ \mathsf{in}\ e \mid [e] \mid \mathsf{let}\ [x] = e\ \mathsf{in}\ e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{nil} \mid \mathsf{cons}(e,e)\mathsf{at}\ e \mid \mathsf{case}\ e\ \mathsf{of\ nil} \to e\ '|'\ \mathsf{cons}(x,y)\mathsf{at}\ z \to e$$

The first line introduces integers, variables, lambda-abstractions, applications, local variable bindings, pairs, and conditionals. The second line introduces recursive definitions, as well as the closedness annotations $[e]$ and $\mathsf{let}\ [x] = e_0\ \mathsf{in}\ e_1$. (We read $[e]$ as "closed of e".)

Intuitively, closedness annotations are assertions reflected in the type and checked by the type system. The first annotation instructs the type checker

[5] While Hofmann's type system for LFPL does not include an explicit typing rule for function definitions, it is essential that functions not introduce new resources when they are applied. A natural way to achieve this, as we show here, is for the body of a function definition to be typed in an environment where only non-linear variables are visible.

to ensure that the value resulting from evaluating e can be safely duplicated, without leading to any runtime heap allocation. If e can be type-checked as such, then the term $[e]$ is assigned a special type. By default, variables will be considered linear. The second annotation allows the programmer to promote the value resulting from evaluating e_0 to a special type environment for non-linear variables, if the type type of e_0 indicates that it can be safely duplicated. The construct then binds this value to x, and makes x available for an arbitrary number of uses in e_1. The syntax for these constructs was inspired by the syntax for S4 modal logic used by Davies and Pfenning [7].

Next in the definition of the syntax are brackets and escapes, which have been described in the previous section. The remaining constructs are constructors and pattern matching over lists. We avoid artificial distinctions between the syntax of terms that will be executed at level 0 and terms executed at level 1. As much as possible, we delegate this distinction to the type system. This simplifies the meta-theory, and allows us to focus on typing issues.

Type System The syntax of types for GeHB is defined as follows:

$$t ::= \mathsf{int} \mid t \to t \mid t * t \mid \diamond \mid [t] \mid \langle t \rangle \mid \mathsf{list}(t)$$

The first three types are standard. The type $\diamond$ (read "diamond") can be intuitively interpreted as the type of the capability for each heap-allocated value. The type $[t]$ (read "closed of t") indicates that a value of that type can be safely duplicated. The semantics for this type constructor was inspired by both the linear type system of LFPL and the closed type constructor used for type-safe reflective and imperative multi-stage programming [24, 18, 4]. The type $\langle \mathrm{t} \rangle$ (read "code of t") is the type of second-stage computations and is associated with all generated LFPL programs. The type $\mathsf{list}(t)$ is an example of a dynamic data structure available in the second stage.

Different types are valid at each level:

$$\frac{}{\overset{n}{\vdash} \mathsf{int}} \qquad \frac{\overset{0}{\vdash} t_0 \quad \overset{0}{\vdash} t_1}{\overset{0}{\vdash} t_0 \to t_1} \qquad \frac{\overset{n}{\vdash} t_0 \quad \overset{n}{\vdash} t_1}{\overset{n}{\vdash} t_0 * t_1} \qquad \frac{}{\overset{1}{\vdash} \diamond} \qquad \frac{\overset{0}{\vdash} t}{\overset{0}{\vdash} [t]} \qquad \frac{\overset{1}{\vdash} t}{\overset{0}{\vdash} \langle t \rangle} \qquad \frac{\overset{1}{\vdash} t}{\overset{1}{\vdash} \mathsf{list}(t)}$$

We will write write t^n for a type t when $\overset{n}{\vdash} t$ is derivable.

Two kinds of typing environments will be used. The first holds values that can be safely duplicated, whereas the second holds linear values. The environments will have the following form:

$$\Gamma ::= \emptyset \mid \Gamma, x : (t^0, 0) \mid \Gamma, x : (t^1 \to t^1, 1) \quad \text{and} \quad \Delta ::= \emptyset \mid \Delta, x : (t^n, n)$$

Thus, all environments carry mappings from variable names to *pairs* of types and binding levels. In Γ, for any given variable binding, if the binding level is 0 then the type can be any type valid at level 0. If the binding level is 1, however, only functions bound at level 1 are allowed. In particular, the only use of Γ at level 1 is to express that functions defined at level 1 (using let rec) can be safely used

$$\frac{}{\Gamma;\Delta \overset{n}{\vdash} i : \mathsf{int}}\ (\text{INT}) \qquad \frac{\Gamma(x) = (t, 0)}{\Gamma;\Delta \overset{0}{\vdash} x : t}\ (\text{VARN})$$

$$\frac{\Delta(x) = (t, n)}{\Gamma;\Delta \overset{n}{\vdash} x : t}\ (\text{VARL}) \qquad \frac{\overset{0}{\vdash} t_0 \quad \Gamma;\Delta, x : (t_0, 0) \overset{0}{\vdash} e : t_1}{\Gamma;\Delta \overset{0}{\vdash} \lambda x.e : t_0 \to t_1}\ (\text{LAM})$$

$$\frac{\Gamma;\Delta_0 \overset{0}{\vdash} e_0 : t_1 \to t_2 \quad \Gamma;\Delta_1 \overset{0}{\vdash} e_1 : t_1}{\Gamma;\Delta_0, \Delta_1 \overset{0}{\vdash} e_0(e_1) : t_2}\ (\text{APP}) \qquad \frac{\Gamma(f) = (t_0 \to t_1, 1) \quad \Gamma;\Delta \overset{1}{\vdash} e : t_0}{\Gamma;\Delta \overset{1}{\vdash} f(e) : t_1}\ (\text{APPF})$$

$$\frac{\Gamma;\Delta_0 \overset{n}{\vdash} e_0 : t_0 \quad \Gamma;\Delta_1, x : (t_0, n) \overset{n}{\vdash} e_1 : t_1}{\Gamma;\Delta_0, \Delta_1 \overset{n}{\vdash} \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 : t_1}\ (\text{LET}) \qquad \frac{\Gamma;\Delta_0 \overset{n}{\vdash} e_0 : \mathsf{int} \quad \Gamma;\Delta_1 \overset{n}{\vdash} e_1 : t \quad \Gamma;\Delta_1 \overset{n}{\vdash} e_2 : t}{\Gamma;\Delta_0, \Delta_1 \overset{n}{\vdash} \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t}\ (\text{IF})$$

$$\frac{\overset{n}{\vdash} t_0, t_1 \quad \Gamma, f : (t_0 \to t_1, n); x : (t_0, n) \overset{n}{\vdash} e_0 : t_1 \quad \Gamma, f : (t_0 \to t_1, n); \Delta \overset{n}{\vdash} e_1 : t_2}{\Gamma;\Delta \overset{n}{\vdash} \mathsf{let\ rec}\ f(x) = e_0\ \mathsf{in}\ e_1 : t_2}\ (\text{REC})$$

$$\frac{\Gamma;\emptyset \overset{0}{\vdash} e : t}{\Gamma;\Delta \overset{0}{\vdash} [e] : [t]}\ (\text{CLOS}) \qquad \frac{\Gamma;\Delta_0 \overset{0}{\vdash} e_0 : [t_0] \quad \Gamma, x : (t_0, 0); \Delta_1 \overset{0}{\vdash} e_1 : t_1}{\Gamma;\Delta_0, \Delta_1 \overset{0}{\vdash} \mathsf{let}\ [x] = e_0\ \mathsf{in}\ e_1 : t_1}\ (\text{LETC})$$

$$\frac{\Gamma;\Delta \overset{n}{\vdash} e : t_1 * t_2}{\Gamma;\Delta \overset{n}{\vdash} \pi_i\ e : t_i}\ (\text{PI}) \qquad \frac{\Gamma;\Delta_0 \overset{n}{\vdash} e_0 : t_0 \quad \Gamma;\Delta_1 \overset{n}{\vdash} e_1 : t_1}{\Gamma;\Delta_0, \Delta_1 \overset{n}{\vdash} (e_0, e_1) : t_0 * t_1}\ (\text{PAIR}) \qquad \frac{\Gamma;\Delta \overset{1}{\vdash} e : t}{\Gamma;\Delta \overset{0}{\vdash} \langle e \rangle : \langle t \rangle}\ (\text{BRAC})$$

$$\frac{\Gamma;\Delta \overset{0}{\vdash} e : \langle t \rangle}{\Gamma;\Delta \overset{1}{\vdash} \tilde{}e : t}\ (\text{ESC}) \qquad \frac{\Gamma;\Delta, x : (t_0, 1), y : (t_0, 1) \overset{1}{\vdash} e : t_1 \quad t_0 \in \{u ::= \mathsf{int} \mid u * u\}}{\Gamma;\Delta, x : (t_0, 1) \overset{1}{\vdash} e[y := x] : t_1}\ (\text{CONTR})$$

$$\frac{\overset{1}{\vdash} t}{\Gamma;\Delta \overset{1}{\vdash} \mathsf{nil} : \mathsf{list}(t)}\ (\text{NIL}) \qquad \frac{\Gamma;\Delta_0 \overset{1}{\vdash} e_0 : t \quad \Gamma;\Delta_1 \overset{1}{\vdash} e_1 : \mathsf{list}(t) \quad \Gamma;\Delta_2 \overset{1}{\vdash} e_2 : \diamond}{\Gamma;\Delta_0, \Delta_1, \Delta_2 \overset{1}{\vdash} \mathsf{cons}(e_0, e_1)\mathsf{at}\ e_2 : \mathsf{list}(t)}\ (\text{CONS})$$

$$\frac{\Gamma;\Delta_0 \overset{1}{\vdash} e_0 : \mathsf{list}(t_0) \quad \Gamma;\Delta_1 \overset{1}{\vdash} e_1 : t_1 \quad \Gamma;\Delta_1, x : (t_0, 1), y : (\mathsf{list}(t_0), 1), l : (\diamond, 1) \overset{1}{\vdash} e_2 : t_1}{\Gamma;\Delta_0, \Delta_1 \overset{1}{\vdash} \mathsf{case}\ e_0\ \mathsf{of\ nil} \to e_1 \mid \mathsf{cons}(x, y)\mathsf{at}\ l \to e_2 : t_1}\ (\text{CASE})$$

Fig. 1. Type System for GeHB

multiple times in the body of a program. For Δ, the situation is straightforward. A valid combined environment $\Gamma;\Delta$ is a pair Γ and Δ such that any variable x either occurs exactly once in one of them or occurs in neither.

We will write Γ^n (and similarly Δ^n) for a Γ where all bindings are at level n.

The judgment $\Gamma;\Delta \overset{n}{\vdash} e : t$ presented in Figure 1 defines the type system for GeHB. The environment Γ is standard whereas the environment Δ is linear. This distinction manifests itself in the type system as a difference in the way these environments are propagated to subterms. As a simple example, consider the rule for application $e_0(e_1)$. The environment Γ is used in type checking both terms. In contrast, the linear environment Δ is split into two parts by pattern matching it with Δ_0 and Δ_1, and exactly one of these two parts is used in type checking each of the subterms e_0 and e_1.

Proposition 1. *1. If* $\Gamma; \Delta \overset{n}{\vdash} e : t$ *is derivable, then so is* $\overset{n}{\vdash} t$.
2. If $\Gamma; \Delta, x : (t^0, 0) \overset{m}{\vdash} e : t'$ *is derivable, then* x *occurs at most once in* e.

The first eight rules are essentially standard. Integers are available at both levels. Two rules are needed for variables, one for the non-linear environment (VARN) and one for the linear environment (VARL). Note that the variable rules require that the binding level be the same as the level at which the variable is used. Additionally, the (VARN) rule only applies at level 0. We can only lookup a level 1 variable bound in Γ if it is used in an application at level 1 (APPF). Lambda abstractions are mostly standard, but they are only allowed at level 0.

The rule for function definitions (REC) makes the newly defined function available as a non-linear variable. To make sure that this is sound, however, we have to require that the body e_0 of this definition be typable using no linear variables except for the function's parameter.

The rule for close (CLOS) uses a similar condition: a closedness annotation around a term e is well typed only if e can be typed without reference to any linear variables introduced by the context. The rule for let-close (LETC) is the elimination rule for the closed type constructor $[t]$. The rule allows us to take any value of closed type and add it to the non-linear environment Γ.

The next two rules for projection and pairing are essentially standard. The rules for brackets and escapes are standard in multi-stage languages. The remaining rules define conditions for terms that are valid only at level 1, and come directly from LFPL.

Lemma 1 (Weakening). *If* $\Gamma \overset{n}{\vdash} e : t$ *then*

1. $\Gamma, x : (t', n'); \Delta \overset{n}{\vdash} e : t$, *and*
2. $\Gamma; \Delta, x : (t', n') \overset{n}{\vdash} e : t$

Lemma 2 (Substitution).

1. *If* $\Gamma; \Delta_0 \overset{0}{\vdash} e_1 : t_1$ *and* $\Gamma; \Delta_1, x : (t_1, 0) \overset{n}{\vdash} e_2 : t_2$ *then* $\Gamma; \Delta_0, \Delta_1 \overset{n}{\vdash} e_2[x := e_1] : t_2$.
2. *If* $\Gamma; \emptyset \overset{0}{\vdash} e_1 : t_1$ *and* $\Gamma, x : (t_1, 0); \Delta_1 \overset{n}{\vdash} e_2 : t_2$ *then* $\Gamma; \Delta_1 \overset{n}{\vdash} e_2[x := e_1] : t_2$.

Operational Semantics for First Stage The judgment $e_0 \overset{n}{\hookrightarrow} e_1$ presented in Figure 2 defines the operational semantics of GeHB programs. Even though the evaluation function used in this paper uses an index n, it is *not* defining how evaluation occurs during both stages. Rather, it defines how the mix of both level 0 and level 1 terms are evaluated *during the first stage*. When the index n is 0 we say we are evaluating e_0, and when the index is 1 we say we are rebuilding this term. Note that this judgment defines only what gets done during the first stage of execution, namely, the generation stage. Execution in the second stage is defined in terms of Hofmann's semantics for LFPL terms (Section 3).

$$\frac{}{i \overset{n}{\hookrightarrow} i} \quad \frac{}{x \overset{1}{\hookrightarrow} x} \quad \frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e} \quad \frac{e_0 \overset{0}{\hookrightarrow} \lambda x.e_2 \quad e_1 \overset{0}{\hookrightarrow} e_3 \quad e_2[x := e_3] \overset{0}{\hookrightarrow} e_4}{e_0(e_1) \overset{0}{\hookrightarrow} e_4}$$

$$\frac{e_0 \overset{1}{\hookrightarrow} e_2 \quad e_1 \overset{1}{\hookrightarrow} e_3}{e_0(e_1) \overset{1}{\hookrightarrow} e_2(e_3)} \quad \frac{e_0 \overset{n}{\hookrightarrow} e_2 \quad e_1 \overset{n}{\hookrightarrow} e_3}{(e_0, e_1) \overset{n}{\hookrightarrow} (e_2, e_3)} \quad \frac{e \overset{0}{\hookrightarrow} (e_1, e_2)}{\pi_i(e) \overset{0}{\hookrightarrow} e_i} \quad \frac{e_0 \overset{1}{\hookrightarrow} e_1}{\pi_i(e_0) \overset{1}{\hookrightarrow} \pi_i(e_1)}$$

$$\frac{e_0 \overset{0}{\hookrightarrow} e_2 \quad e_1[x := e_2] \overset{0}{\hookrightarrow} e_3}{\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \overset{0}{\hookrightarrow} e_3} \quad \frac{e_0 \overset{1}{\hookrightarrow} e_2 \quad e_1 \overset{1}{\hookrightarrow} e_3}{\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \overset{1}{\hookrightarrow} \mathsf{let}\ x = e_2\ \mathsf{in}\ e_3}$$

$$\frac{e_0 \overset{0}{\hookrightarrow} i \quad i \neq 0 \quad e_1 \overset{0}{\hookrightarrow} e_3}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \overset{0}{\hookrightarrow} e_3} \quad \frac{e_0 \overset{0}{\hookrightarrow} 0 \quad e_2 \overset{0}{\hookrightarrow} e_3}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \overset{0}{\hookrightarrow} e_3}$$

$$\frac{e_0 \overset{1}{\hookrightarrow} e_3 \quad e_1 \overset{1}{\hookrightarrow} e_4 \quad e_2 \overset{1}{\hookrightarrow} e_5}{\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \overset{1}{\hookrightarrow} \mathsf{if}\ e_3\ \mathsf{then}\ e_4\ \mathsf{else}\ e_5} \quad \frac{e_1[f := \mathsf{let\ rec}\ f(x) = e_0\ \mathsf{in}\ \lambda x.e_0] \overset{0}{\hookrightarrow} e_2}{\mathsf{let\ rec}\ f(x) = e_0\ \mathsf{in}\ e_1 \overset{0}{\hookrightarrow} e_2}$$

$$\frac{e_0 \overset{1}{\hookrightarrow} e_2 \quad e_1 \overset{1}{\hookrightarrow} e_3}{\mathsf{let\ rec}\ f(x) = e_0\ \mathsf{in}\ e_1 \overset{1}{\hookrightarrow} \mathsf{let\ rec}\ f(x) = e_2\ \mathsf{in}\ e_3} \quad \frac{e_0 \overset{n}{\hookrightarrow} e_1}{[e_0] \overset{n}{\hookrightarrow} [e_1]}$$

$$\frac{e_0 \overset{0}{\hookrightarrow} [e_3] \quad e_1[x := e_3] \overset{0}{\hookrightarrow} e_2}{\mathsf{let}\ [x] = e_0\ \mathsf{in}\ e_1 \overset{0}{\hookrightarrow} e_2} \quad \frac{e_0 \overset{1}{\hookrightarrow} e_2 \quad e_1 \overset{1}{\hookrightarrow} e_3}{\mathsf{let}\ [x] = e_0\ \mathsf{in}\ e_1 \overset{1}{\hookrightarrow} \mathsf{let}\ [x] = e_2\ \mathsf{in}\ e_3}$$

$$\frac{e_0 \overset{1}{\hookrightarrow} e_1}{\langle e_0 \rangle \overset{0}{\hookrightarrow} \langle e_1 \rangle} \quad \frac{e_0 \overset{0}{\hookrightarrow} \langle e_1 \rangle}{\tilde{}e_0 \overset{1}{\hookrightarrow} e_1} \quad \frac{}{\mathsf{nil} \overset{1}{\hookrightarrow} \mathsf{nil}} \quad \frac{e_0 \overset{1}{\hookrightarrow} e_3 \quad e_1 \overset{1}{\hookrightarrow} e_4 \quad e_2 \overset{1}{\hookrightarrow} e_5}{\mathsf{cons}(e_0, e_1)\mathsf{at}\ e_2 \overset{1}{\hookrightarrow} \mathsf{cons}(e_3, e_4)\mathsf{at}\ e_5}$$

$$\frac{e_0 \overset{1}{\hookrightarrow} e_3 \quad e_1 \overset{1}{\hookrightarrow} e_4 \quad e_2 \overset{1}{\hookrightarrow} e_5}{\mathsf{case}\ e_0\ \mathsf{of} \begin{cases} \mathsf{nil} \to e_1 \\ \text{'|'}\ \mathsf{cons}(x, y)\mathsf{at}\ z \to e_2 \end{cases} \overset{1}{\hookrightarrow} \mathsf{case}\ e_3\ \mathsf{of} \begin{cases} \mathsf{nil} \to e_4 \\ \text{'|'}\ \mathsf{cons}(x, y)\mathsf{at}\ z \to e_5 \end{cases}}$$

Fig. 2. Operational Semantics of First Stage in GeHB

Lemma 3 (Type Preservation). *If* $\Gamma^1, \Delta^1 \overset{n}{\vdash} e : t$ *and* $e \overset{n}{\hookrightarrow} e'$ *then* $\Gamma^1, \Delta^1 \overset{n}{\vdash} e' : t$.

Generated Programs as LFPL Programs Hofmann's type system does not have an explicit rule for recursive definitions. Instead, the system assumes a signature of top-level functions under which programs are typable. It remains for us to show that programs generated by GeHB can be converted into LFPL programs. The key observations needed to make this connection are as follows:

1. A code value $\langle e \rangle$ generated by evaluating a GeHB program is free of escapes and level 0 terms.
2. The typing of let rec ensures that all such declarations can be lifted to top-level.

3. The result of lifting is a sequence of function declarations ending with a proper LFPL term.

In what follows we justify these claims.

Code Values are Escape Free To establish this property it is simpler to work with a superset of typed terms called expression families [24, 26]. Expression families classify terms based on appropriateness for appearing at a certain level:

$$\begin{aligned} e^n \in \mathbb{E}^n ::= {} & i \mid x \mid e^n(e^n) \mid \mathsf{let}\ x = e^n\ \mathsf{in}\ e^n \mid (e^n, e^n) \mid \pi_1\ e^n \mid \pi_2\ e^n \mid \\ & \mathsf{if}\ e^n\ \mathsf{then}\ e^n\ \mathsf{else}\ e^n \mid \mathsf{let\ rec}\ f(x) = e^n\ \mathsf{in}\ e^n \\ e^0 \in \mathbb{E}^0 \mathrel{+}= {} & \lambda x.e^0 \mid [e^0] \mid \mathsf{let}\ [x] = e^0\ \mathsf{in}\ e^0 \mid \langle e^1 \rangle \\ e^1 \in \mathbb{E}^1 \mathrel{+}= {} & \tilde{}e^0 \mid \mathsf{nil} \mid \mathsf{cons}(e^1, e^1)\mathsf{at}\ e^1 \mid \mathsf{case}\ e^1 \mathsf{of\ nil} \to e^1\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to e^1 \end{aligned}$$

The first line defines expressions that can be associated with either level. The second and third definitions extend the set of terms that can be associated with levels 0 and 1, respectively. Values are defined similarly:

$$\begin{aligned} v \in \mathbb{V}^0 ::= {} & i \mid \lambda x.e^0 \mid (v, v) \mid [v] \mid \langle g \rangle \\ g \in \mathbb{V}^1 ::= {} & i \mid x \mid g(g) \mid \mathsf{let}\ x = g\ \mathsf{in}\ g \mid (g, g) \mid \pi_1\ g \mid \pi_2\ g \mid \\ & \mathsf{if}\ g\ \mathsf{then}\ g\ \mathsf{else}\ g \mid \mathsf{let\ rec}\ f(x) = g\ \mathsf{in}\ g \\ & \mathsf{nil} \mid \mathsf{cons}(g, g)\mathsf{at}\ g \mid \mathsf{case}\ g\ \mathsf{of\ nil} \to g\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to g \end{aligned}$$

For level 0 values, the first three cases are standard. Values with the closed code constructor at the head must carry a level 0 value (thus $[t]$ is a strict constructor). Code values must carry a subterm that is a level 1 value. Level 1 values are level 1 expressions that do not contain escapes.

Proposition 2. *1.* $\mathbb{V}^n \subseteq \mathbb{E}^n$*, 2.* $\Gamma; \Delta \overset{n}{\vdash} e : t$ *implies* $e \in \mathbb{E}^n$*, and 3.* $e^n \overset{n}{\hookrightarrow} e'$ *implies* $e' \in \mathbb{V}^n$*.*

Function Declarations can be Lifted The subset of GeHB terms corresponding to LFPL terms is easily defined as the following subset of $\mathbb{V}^1$: [6]

$$\begin{aligned} h \in \mathbb{H} ::= {} & i \mid x \mid h(h) \mid \mathsf{let}\ x = h\ \mathsf{in}\ h \mid (h, h) \mid \pi_1\ h \mid \pi_2\ h \mid \mathsf{if}\ h\ \mathsf{then}\ h\ \mathsf{else}\ h \\ & \mathsf{nil} \mid \mathsf{cons}(h, h)\mathsf{at}\ h \mid \mathsf{case}\ h\ \mathsf{of\ nil} \to h\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to h \end{aligned}$$

A term h is typable in our system as $f_i : (t_i^1 \to t_i'^1, 1); \emptyset \overset{1}{\vdash} h : t$ if and only if it is typable as $\emptyset \vdash h : t$ in LFPL's type system under a function signature $f_i : t_i^1 \to t_i'^1$.

Lifting recursive functions can be performed by a sequence of reductions carried out in the appropriate context. Declaration contexts are defined as follows:

$$D \in \mathbb{D} ::= [] \mid \mathsf{let\ rec}\ f(x) = D[h]\ \mathsf{in}\ D$$

[6] The LFPL calculus has no explicit let construct, but let has no effect on heap usage.

We will show that for every term g there is a unique term of the form $D[h]$. The essence of the argument is the following partial function from $\mathbb{V}^1$ to $\mathbb{V}^1$:

$$\frac{}{i \mapsto i} \quad \frac{}{x \mapsto x} \quad \frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2]}{g_1(g_2) \mapsto D_1[D_2[h_1(h_2)]]}$$

$$\frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2] \quad x \notin FV(D_2)}{\mathsf{let}\ x = g_1\ \mathsf{in}\ g_2 \mapsto D_1[D_2[\mathsf{let}\ x = h_1\ \mathsf{in}\ h_2]]} \quad \frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2]}{(g_1, g_2) \mapsto D_1[D_2[(h_1, h_2)]]}$$

$$\frac{g \mapsto D[h]}{\pi_i\ g \mapsto D[\pi_i\ h]} \quad \frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2] \quad g_3 \mapsto D_3[h_3]}{\mathsf{if}\ g_1\ \mathsf{then}\ g_2\ \mathsf{else}\ g_3 \mapsto D_1[D_2[D_3[\mathsf{if}\ h_1\ \mathsf{then}\ h_2\ \mathsf{else}\ h_3]]]}$$

$$\frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2]}{\mathsf{let\ rec}\ f(x) = g_1\ \mathsf{in}\ g_2 \mapsto \mathsf{let\ rec}\ f(x) = D_1[h_1]\ \mathsf{in}\ D_2[h_2]}$$

$$\frac{}{\mathsf{nil} \mapsto \mathsf{nil}} \quad \frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2] \quad g_3 \mapsto D_3[h_3]}{\mathsf{cons}(g_1, g_2)\mathsf{at}\ g_3 \mapsto D_1[D_2[D_3[\mathsf{cons}(h_1, h_2)\mathsf{at}\ h_3]]]}$$

$$\frac{g_1 \mapsto D_1[h_1] \quad g_2 \mapsto D_2[h_2] \quad g_3 \mapsto D_3[h_3] \quad x, y, z \notin FV(D_2)}{\begin{array}{l}\mathsf{case}\ g_1\ \mathsf{of\ nil} \to g_2\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to g_3 \\ \mapsto D_1[D_2[D_3[\mathsf{case}\ h_1\ \mathsf{of\ nil} \to h_2\ '|'\ \mathsf{cons}(x, y)\mathsf{at}\ z \to h_3]]]\end{array}}$$

The definition above uses the Barendregt convention [1] of assuming that all bound variables are distinct in terms such as $D_1[h_1]$ and $D_2[h_2]$. This allows us to construct terms such as $D_1[D_2[h_1(h_2)]]$ without introducing accidental variable capture.

Proposition 3. *The following properties capture the basic features of lifting and declaration contexts:*

1. *$g \mapsto e$ implies that there is a unique pair D and h such that $D[h] = e$.*
2. *$\Gamma; \Delta \overset{1}{\vdash} D[e] : t$ implies that there exists a $\Gamma' = f_i : (t_i^1 \to t_i'^1, 1)$ such that $\Gamma, \Gamma'; \Delta \overset{1}{\vdash} e : t$. Furthermore, if $\Gamma, \Gamma'; \Delta \overset{1}{\vdash} e' : t$, then $\Gamma; \Delta \overset{1}{\vdash} D[e'] : t$. Finally, $x \in dom(\Delta)$ implies $x \notin FV(D)$.*
3. *$\Gamma; \Delta \overset{1}{\vdash} g : t$ and $g \mapsto e$ implies $\Gamma; \Delta \overset{1}{\vdash} e : t$. (Type-preservation for lifting.)*
4. *$\Gamma; \Delta \overset{1}{\vdash} g : t$ implies there is an e such that $g \mapsto e$. (Totality for lifting.)*

Proof. Part 1 is by a simple induction over the lifting derivation. Part 2 is by induction over the structure of D, and noting that let recs only extend Γ. Part 3 is by induction over the height of the lifting derivation, using the first fact in Part 2 as well as weakening. Part 4 is by induction over the typing derivation, and noting that the only risk to totality is the side conditions on free variables. These side conditions only arise with binding constructs, namely, let and case. For both constructs, the last observation in Part 2 implies that these side conditions hold.

Theorem 1 (Generated Programs are LFPL Programs).
Whenever $\emptyset; x_j : (\diamond, 1) \overset{0}{\vdash} e : \langle t \rangle$ and $e \overset{0}{\hookrightarrow} e'$, then

– *there exists a term* g *such that* $\langle g \rangle = e'$,
– *there exists* D *and* h *such that* $g \mapsto D[h]$,
– *there exists* $\Gamma' = f_i : (t_i^1 \to t_i'^1, 1)$ *such that* $\Gamma'; x_j : (\diamond, 1) \overset{1}{\vdash} h : t$, *and*
– *the LFPL judgment* $x_j : \diamond \vdash h : t$ *is derivable under signature* $f_i : t_i^1 \to t_i'^1$.

4 Parameterized Insertion Sort in GeHB

GeHB allows us to parameterize the insertion sort function presented in the introduction with a comparison operator. This generalization takes advantage of both staging and closedness annotations. Intuitively, this function will take as argument a staged comparison function, a second-stage list computation, and returns a second-stage list computation. Formally, the type of such a function would be:

```
(<A>*<A> -> <int>) * <list(A)> -> <list(A)>
```

for any given type `A`. But because this function will itself be a generator of function declarations, and the body of a function declaration cannot refer to any linear variables except for its arguments, we will need to use closedness annotations to express the fact that we have more information about the comparison operator. In particular, we *require* that the comparison operation not be allowed to allocate second-stage heap resources. The closed type constructor `[...]` allows us to express this requirement in the type of the generalized sort function as follows:

```
[<A>*<A> -> <int>] * <list(A)> -> <list(A)>
```

The staged parameterized sort function itself can be defined as follows:

```
let rec sort_gen(f,ll) =
  let [f'] = f in
  <let rec insert(d,a,l) =
     case l of
       nil -> cons(a, nil) at d
     | cons(b,t) at d' -> if ~(f'(<a>, <b>))
                          then cons(a, cons(b, t) at d') at d
                          else cons(b, insert(a, d', t)) at d
   in let rec sort(l) =
        case l of
          nil -> nil
        | cons(a,t) at d -> insert(d, a, sort(t))
      in sort(~ll)>

in sort_gen([fun (x,y) -> <fst(~x) > fst(~y)>],
            <cons((3,33), cons((100,12), cons((44,100),
               cons((5,13),nil) at d4) at d3) at d2) at d1>)
```

Without the staging and the closedness annotations, this definition is standard. We assume that we are given four free heap locations `d1` to `d4` to work with. As illustrated in Section 2, the staging annotations separate first stage computations from second stage ones. Closedness annotations appear in two places in the program. Once on the second line of the program, and once around the first argument to the `sort_gen` function at the end of the program. The closedness annotation at the end of the program asserts that the program fragment `fun (x,y) -> <fst(~x) > fst(~y)>` is free of operations that allocate resources on the heap during the second stage. Because function parameters are linear, and function definitions can only use their formal parameters, the variable `f` cannot be used in the body of the (inner) declaration of `insert`. Knowing that this variable has closed type, however, allows us to use the construction `let [f']=f` to copy the value of `f` to the non-linear variable `f'`, which can thereafter be used in the body of `insert`.

Evaluating the GeHB program above generates the following code fragment:

```
<let rec insert(d,a,l) =
  case l of nil -> cons(a, nil) at d
          | cons(b,t) at d' -> if fst(a) > fst(b)))
                               then cons(a, cons(b, t) at d') at d
                               else cons(b, insert(a, d', t)) at d
 in let rec sort(l) =
     case l of nil -> nil
             | cons(a,t) at d -> insert(d, a, sort(t))

    in sort(cons((3,33), cons((100,12), cons((44,100),
              cons((5,13),nil) at d4) at d3) at d2) at d1)>
```

The generated program is a valid LFPL program, and in this particular case, no lifting of function declarations is required. Note also that because higher-order functions are only present in the first stage, there is no runtime cost for parameterizing `insert` with respect to the comparison operator `f`. In particular, the body of the comparison operation will always be inlined in the generated sorting function.

5 Discussion

An unexpected finding from the study of GeHB is that generating LFPL programs requires the use of a primarily linear type system in the first stage. This is achieved using a single type constructor $[t]$ that captures both linearity [3] *and* closedness [4, 24]. Both notions can separately be interpreted as comonads (c.f. [3] and [2]). The type constructor $[t]$ seems novel in that it compacts two comonads into one constructor. From a language design perspective, the possibility of combining two such comonads into one is attractive, because it reduces annotations associated with typing programs that use these comonads. A natural and important question to ask is whether the compactness of $[t]$ is an accident or an instance of a more general pattern. It will be interesting to see whether the useful composability of monads translates into composability of comonads.

GeHB provides new insights about multi-stage languages. In particular, we find that the closed type constructor is useful even when the language does not support reflection (the `run` construct). Because the notion of closedness is self-motivated in GeHB, we are hopeful that it will be easy to introduce a construct like `run` to this setting. With such a construct, not only will the programmer be able to construct computations for an embedded platform, but also to execute and receive results of such computations. An important practical problem for general multi-stage programming is code duplication. In the language presented here, no open code fragments can be duplicated. It will be interesting to see if, in practice, a linear typing approach could also be useful to address code duplication in multi-stage languages.

Czarnecki et al. [6] describe a template-based system for generating embedded software for satellite systems. In this approach programs are generated automatically from XML specifications and domain-specific libraries written in languages like Ada and Fortran. Their system was designed with the explicit goal of producing human-readable code and relies on the programmer to verify its correctness. In contrast, our goal is to design languages that let programmers reason about generated programs while they are reading and writing *the generator*. If enough guarantees can be made just by type checking the generator, then we know that the full *family* of programs produced by this generator is well-behaved. The contributions of GeHB and Czarnecki et al. are orthogonal and compatible.

6 Conclusions and Future Work

We have presented the two-level language GeHB that offers more expressivity than LFPL and at the same time offers more resource guarantees than general-purpose multi-stage languages. The technical challenge in designing GeHB lies in finding the appropriate type system. The development is largely modular, in the sense that many results on LFPL can be reused. While we find that a naive combination of two-level languages and LFPL is not satisfactory, we are able to show that recently proposed techniques for type-safe, reflective, and imperative multi-stage programming can be used to overcome these problems.

Future work will focus on studying the applicability of two-level languages for other kinds of resource bounds. For example, languages such as RT-FRP [28] and E-FRP [29] have complete bounds on all runtime space allocation. This is achieved by interpreting recursive definitions as difference equations indexed by the sequence of events or stimuli that drive the system. An important direction for future work is to define a useful class of terminating recursive definitions with a standard interpretation. Three interesting approaches appear in the literature, which we plan to investigate in future work: The use of a type system that explicitly encodes an induction principle, which allows the programmer to use recursion, as long as the type system can check that it is well-founded [12, 11], the use of special iterators that are always guaranteed to terminate [5], enriching all types with information about space needed to store values of the respective types, and the use of the principle of non-size-increasing parameters [16, 15].

Acknowledgements: Eric Allen and Jonathan Bannet read and commented on early drafts. We would also like to thank Paul Hudak and Valery Trifonov for discussions when these ideas were in their formative stages.

References

1. Barendregt, H. P. Lambda calculi with types. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, Oxford, 1991.
2. Benaissa, Z. E.-A., Moggi, E., Taha, W., and Sheard, T. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)* (1999).
3. Benton, N., and Wadler, P. Linear logic, monads and the lambda calculus. In *the Symposium on Logic in Computer Science (LICS '96)* (New Brunswick, 1996), IEEE Computer Society Press.
4. Calcagno, C., Moggi, E., and Taha, W. Closed types as a simple approach to safe imperative multi-stage programming. In *the International Colloquium on Automata, Languages, and Programming (ICALP '00)* (Geneva, 2000), vol. 1853 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 25–36.
5. Crary, K., and Weirich, S. Resource bound certification. In *the Symposium on Principles of Programming Languages (POPL '00)* (N.Y., Jan. 19–21 2000), ACM Press, pp. 184–198.
6. Czarnecki, K., Bednasch, T., Unger, P., and Eisenecker, U. Generative programming for embedded software: An industrial experience report. In *Generative Programming and Component Engineer SIGPLAN/SIGSOFT Conference, GPCE 2002* (Oct. 2002), D. Batory, C. Consel, and W. Taha, Eds., vol. 2487 of *Lecture Notes in Computer Science*, ACM, Springer, pp. 156–172.
7. Davies, R., and Pfenning, F. A modal analysis of staged computation. *Journal of the ACM 48*, 3 (2001), 555–604.
8. Hofmann, M. Linear types and non-size-increasing polynomial time computation. In *the Symposium on Logic in Computer Science (LICS '99)* (July 1999), IEEE, pp. 464–473.
9. Hofmann, M. A type system for bounded space and functional in-place update. *Nordic Journal of Computing 7*, 4 (Winter 2000).
10. Hofmann, M. A type system for bounded space and functional in-place update. In *European Symposium on Programming (ESOP)* (2000), Lecture Notes in Computer Science, Springer-Verlag.
11. Hughes, R., and Pareto, L. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99)* (N.Y., Sept. 27–29 1999), vol. 34.9 of *ACM Sigplan Notices*, ACM Press, pp. 70–81.
12. Hughes, R., Pareto, L., and Sabry, A. Proving the correctness of reactive systems using sized types. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (St Petersburg, Florida, 1996), G. L. S. Jr, Ed., vol. 23, ACM Press.
13. Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
14. Kamin, S., Callahan, M., and Clausen, L. Lightweight and generative components II: Binary-level components. In *[25]* (2000), pp. 28–50.

15. Lee, C. S. Program termination analysis in polynomial time. In *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002* (Oct. 2002), D. Batory, C. Consel, and W. Taha, Eds., vol. 2487 of *Lecture Notes in Computer Science*, ACM, Springer, pp. 218–235.
16. Lee, C. S., Jones, N. D., and Ben-Amram, A. M. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages* (january 2001), vol. 28, ACM press, pp. 81–92.
17. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from http://www.cs.rice.edu/ taha/MetaOCaml/, 2001.
18. Moggi, E., Taha, W., Benaissa, Z. E.-A., and Sheard, T. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)* (1999), vol. 1576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 193–207.
19. Mycroft, A., and Sharp, R. A statically allocated parallel functional language. In *Automata, Languages and Programming* (2000), pp. 37–48.
20. National Instruments. Introduction to LabVIEW Real-Time. Available online from http://volt.ni.com/niwc/labviewrt/lvrt_intro.jsp?node=2381&node=2381, 2003.
21. National Instruments. LabVIEW FPGA Module. Available online from http://sine.ni.com/apps/we/nioc.vp?cid=11784&lang=US, 2003.
22. Nielson, F., and Nielson, H. R. *Two-Level Functional Languages*. No. 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.
23. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html.
24. Taha, W. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [23].
25. Taha, W., Ed. *Semantics, Applications, and Implementation of Program Generation* (Montréal, 2000), vol. 1924 of *Lecture Notes in Computer Science*, Springer-Verlag.
26. Taha, W. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM)* (Boston, 2000), ACM Press.
27. Taha, W., Hudak, P., and Wan, Z. Directions in functional programming for real(-time) applications. In *the International Workshop on Embedded Software (EMSOFT '01)* (Lake Tahoe, 2001), vol. 221 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 185–203.
28. Wan, Z., Taha, W., and Hudak, P. Real-time FRP. In *the International Conference on Functional Programming (ICFP '01)* (Florence, Italy, September 2001), ACM.
29. Wan, Z., Taha, W., and Hudak, P. Event-driven FRP. In *Proceedings of Fourth International Symposium on Practical Aspects of Declarative Languages* (Jan 2002), ACM.