# Implicitly Heterogeneous Multi-Stage Programming

J. ECKHARDT, R. KAIABACHEV, E. PASALIC, K. SWADI, and W. TAHA

*Rice University*
*Department of Computer Science*
*P.O. Box 1892, MS 132*
*Houston, TX 77005, USA*

{jle,roumen,pasalic,kswadi,taha}@cs.rice.edu

Received 30 August 2005

***Abstract*** Previous work on semantics-based multi-stage programming (MSP) language design focused on *homogeneous* designs, where the generating and the generated languages are the same. Homogeneous designs simply add a hygienic quasi-quotation and evaluation mechanism to a base language. An apparent disadvantage of this approach is that the programmer is bound to both the expressivity and performance characteristics of the base language. This paper proposes a practical means to avoid this by providing specialized translations from subsets of the base language to different target languages. This approach preserves the homogeneous "look" of multi-stage programs, and, more importantly, the static guarantees about the generated code. In addition, compared to an explicitly heterogeneous approach, it promotes reuse of generator source code and systematic exploration of the performance characteristics of the target languages.

Supported by NSF SoD-0439017 "Synthesizing Device Drivers", NSF ITR-0113569 "Putting Multi-Stage Annotations to Work", Texas ATP 003604-0032-2003 "Advanced Languages Techniques for Device Drivers" and NSF ITR-0205303 "Building Practical Compilers Based on Adaptive Search."

This is a revised version based on a conference paper of the same title [4)].

To illustrate the proposed approach, we design and implement a translation to a subset of C suitable for numerical computation, and show that it preserves static typing. The translation is implemented, and evaluated with several benchmarks. The implementation is available in the online distribution of MetaOCaml.

**Keywords** Functional Programming, Multi-Stage Programming.

# §1 Introduction

Multi-stage programming (MSP) languages allow the programmer to use abstraction mechanisms such as functions, objects, and modules, without having to pay a runtime overhead for them due to the generation of specialized code. Operationally, these languages provide quasi-quotation and `eval` mechanisms similar to those of LISP and Scheme. In addition, to avoid accidental capture, bound variables are always renamed, and values produced by quasi-quotes can only be de-constructed by `eval`. This makes reasoning about quoted terms as programs sound [18], even in the untyped setting. Several type systems have been developed that statically ensure that all programs generated using these constructs are well-typed (see for example [19, 1]).

Currently, the main examples of MSP languages include MetaScheme [7], MetaML [20], MetaOCaml [10], and Metaphor [11]. They are based, respectively, on Scheme, Standard ML, OCaml, and Java/C#. In all these cases, the language design is *homogeneous*, in that quoted values are fragments of the base language. Homogeneity has three distinct advantages. First, it is convenient for the language designer, as it often reduces the size of the definitions needed to model a language, and makes extensions to arbitrary stages feasible at little cost. Second, it is convenient for the language implementor, as it allows the implementation of the base language to be reused: In all three examples above, as in LISP and Scheme implementations, the `eval`-like construct calls the underlying implementation. In the case of MetaOCaml, the MetaOCaml compiler and the bytecode runtime system can be used to execute generated programs at runtime. Third, it is convenient for the programmer, as it requires learning only a small number of new language constructs.

While the homogeneous approach described above has its advantages, there are situations where the programmer may wish to take advantage of the capabilities of other compilers that are only available for other languages. For example, very well-developed, specialized compilers exist for application domains

such as numerical computing, embedded systems, and parallel computation.

At first glance, this situation might suggest a need for *heterogeneous* quotation mechanisms, where quoted terms can contain expressions in a different language. Indeed, this approach has been used successfully for applications such as light-weight components [8, 9], FFT [6], and computer graphics [5]. But a heterogeneous quotation mechanism also introduces two new complications:

1. How do we ensure that the generated program is statically typed?
2. How do we avoid restricting a generator to a particular target language?

One approach to addressing the first issue is to develop specialized two-level type systems. This means the language designer must work with type systems and semantics that are as big as both languages combined. Another possibility is to extend meta-programming languages with dependent type systems [13] and thus give the programmers the ability to write data-types that encode the abstract syntax of only *well-typed* object-language terms. Currently, such type systems can introduce significant notational overhead for the programmer, as well as requiring familiarity with type systems that are not yet available in mainstream languages.

In principle, the second problem can be avoided by parameterizing the generators themselves by constructs for the target language of choice. However, this is likely to reduce the readability of the generators, and it is not clear how a quotation mechanism can be used in this setting. Furthermore, to ensure that the generated program is statically typed, we would, in essence, need to parameterize the static type of the generator by a description of the static type system of the target language.

GADTs are beginning to gain increasing acceptance for statically encoding object-language type systems and for statically checking program generators [22, 23, 24, 25]. However, writing type-preserving program generators with GADTs can be awkward: absence of quotation mechanisms makes programs difficult to read; the encodings of object-language type systems tend to be awkward and elaborate (e.g., using de Bruijn indexes for variables, explicit substitutions for manipulating binding constructs). Currently, this kind of typing infrastructure needed is likely to be beyond what has gained acceptance in mainstream programming.

## 1.1 Contributions

This paper proposes a practical approach to avoiding the two problems, which can be described as *implicitly heterogeneous* MSP. In this approach, the programmer does not need to know about the details of the target-language representation. These details are addressed by the meta-language designer once and for all and invoked by the programmer through the familiar interface used to execute generated code. The language implementer provides specialized translations from subsets of the base language to different target languages. Thus, the homogeneous "look" of homogeneous MSP is preserved. An immediate benefit is that the programmer may not have to make any changes to existing generators to target different languages. Additionally, if the translation itself is type preserving, the static guarantee about the type correctness of generated code is maintained.

The proposed approach is studied in the case when the target language is C. After a brief introduction to MSP (Section 2), we outline the details of what can be described as an *offshoring* translation that we have designed and implemented. Designing the translation begins by identifying the precise subset of the target language that we wish to make available to the programmer (Section 3). Once that is done, the next challenge is to identify an appropriate subset in the base language that can be used to represent the target subset.

Like a compiler, offshoring translations are provided by the language implementor and not by the programmer. But the requirements on offshoring translators are essentially the opposite of those on compilers (or compiling translators, such as Tarditi's [21]): First, offshoring is primarily concerned with the target, not the source language. The most expressive translation would cover the full target language, but not necessarily the source language. In contrast, a compiler must cover the source but not necessarily the target language. Second, the translation must be a direct mapping from source to target, and not a complex, optimizing translation. The direct connection between the base and target representations is essential for giving the programmer access to the target language.

To ensure that all generated programs are well typed, we show that the offshoring translation is type preserving (Section 4). Again, this is something the language designer does once and benefits any programmer who uses the offshoring translation.

Having an offshoring translation makes it easier for the programmer to

experiment with executing programs either in OCaml or C (using different C compilers). We present a detailed analysis of changes in performance behavior for a benchmark of dynamic programming algorithms (Section 5). Not surprisingly, C consistently outperforms the OCaml bytecode compiler. We also find that in several cases, the overhead of marshalling the results from the OCaml environment to the C environment and back can be a significant bottleneck, especially in cases where C is much faster than the OCaml bytecode compiler. And while C outperforms the OCaml native code compiler in many cases, there are exceptions.

Postscript: The key features of multi-stage programming (MSP) languages are that they 1) support statically-typed, hygienic quasi-quotations, and 2) that they have a run construct. While the name suggests an emphasis on multi-level computation (rather than just two-level computation), this is not the intention. Its proper use is to refer to writing staged programs in a language that has the two above mentioned properties. This paper is concerned primarily with offshoring in a two-level setting.

## §2 Multi-Stage Programming

MSP languages [20, 16] provide three high-level constructs that allow the programmer to divide computations into distinct stages. These constructs can be used to construct, combine, and execute code fragments. Standard problems associated with the manipulation of code fragments, such as accidental variable capture and the representation of programs, are hidden from the programmer (see for example [16]). The following minimal example illustrates MSP programming in MetaOCaml:

```
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 constructs (brackets `.<`$e$`>.`, escapes `.~`$e$, as well as run `.!` $e$) 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 simply returns a function that would invoke the `power` function every time it gets invoked with a value for $x$. In contrast, the staged version builds a function that computes the third power directly (that is, using only multiplication). To see how the staging constructs work, we can start from the last statement in the code above. Whereas a term `fun x ->`$e$ `x` is a value, an annotated term `.<fun x ->` `.~(`$e$ `.<x>.)>.` is not, because the outer brackets contain an escaped expression

that still needs to be evaluated. Brackets mean that we want to construct a future stage computation, and escapes mean that we want to perform an immediate computation *while* building the bracketed computation. In a multi-stage language, these constructs are not hints, they are imperatives. Thus, the application $e$ `.<x>.` must be performed even though `x` is still an uninstantiated symbol. The expression `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` first results in the equivalent of

```
.! .<fun x -> x*x*x*1>.
```

Once the argument to run (`.!`) is a code fragment that has no escapes, it is compiled and evaluated, and returns a function that has the same performance as if we had explicitly coded the last declaration as:

```
let power3 = fun x -> x*x*x*1
```

Applying this function does not incur the unnecessary overhead that the unstaged version would have had to pay every time `power3` is used.

# §3 Offshoring for Numerical Computation in C

This section presents an example of an offshoring translation aimed at supporting implicitly heterogeneous MSP for basic numerical computation. The method for defining the types and the syntax is presented, along with the resulting formal definitions. It bears repeating that the programmer will not need to write her program generators to explicitly produce the target language presented in this section. Rather, the interface to generating target programs is written using the source-language syntax (Section 3.3); however, the programmer does need to be aware of the target subset definitions to be able to express her algorithms in a way that makes the generated code amenable to translation. (Translation is formally defined in Section 3.5.) This section concludes by presenting the details of the programmer's interface to offshoring, and discussing the practical issue of marshalling values between the runtime systems of the base and target languages.

## 3.1 Types for Target Subset

The types in the target C subset include:

1. Base numerical types int, double and char.
2. One- and two- dimensional arrays of these numerical types. One-dimensional arrays are represented as C arrays. Two-dimensional arrays are implemented as an array of pointers (C type **[]). While this representation is less efficient than two-dimensional arrays in C, it was chosen because it allows for a simpler translation, since it is a better semantic match with OCaml arrays. OCaml array *types* do not explicitly declare their size, so it is not always possible to obtain a valid two-dimensional array type in C from a type of two-dimensional arrays in OCaml.
3. Functions that take base types or arrays of base types as arguments and return base type values. This subset does not include function pointers, functions with variable number of arguments, or functions that return void.

The C subset types are described by the following BNF:

$$\begin{array}{llll} \textit{Base types} & b & \in & \{\mathsf{int}, \mathsf{double}, \mathsf{char}\} \\ \textit{Array types} & a & ::= & b\,[\,] \mid *\,b\,[\,] \\ \textit{Types} & t & ::= & b \mid a \\ \textit{Funtypes} & f & ::= & b\,(t_0, \ldots, t_n) \end{array}$$

## 3.2 Syntax for Target Subset

Figure 1 presents the BNF of the target C subset. The set is essentially a subset of C that has all the basic numerical and array operations, first order functions, and structured control flow operators. The target subset is not determined in a vacuum but also involves considering what can be expressed naturally in the source language. The main restrictions we impose on this subset are:

1. All declarations are initialized. This restriction reflects the fact that OCaml does not permit uninitialized bindings for variables, and representing uninstantiated declarations in OCaml can add complexity.
2. No unstructured control flow statements (i.e., goto, break, continue and fall-through non-defaulted switch statements). This restriction is motivated by the lack of equivalent unstructured control flow operators in OCaml. For the same reason, the increment operations are also limited.

| | | | |
|---|---|---|---|
| *Constant* | $c$ | $\in$ | $Int \cup Float \cup Char$ |
| *Variable* | $x$ | $\in$ | $X$ |
| *Type keyword* | $t$ | ::= | int \| double \| char |
| *Declaration* | $d$ | ::= | $t\ x = c \mid t\ x[n] = \{c^*\} \mid t\ \texttt{*}x[n] = \{x^*\}$ |
| *Arguments* | $a$ | ::= | $t\ x \mid t\ x[\,] \mid t\ \texttt{*}x[\,]$ |
| *Unary operator* | $f^{(1)}$ | ::= | (float) \| (int) \| cos \| sin \| sqrt |
| *Binary operator* | $f^{[2]}$ | ::= | + \| - \| * \| / \| % \| && \| \|\| \| & \| \| \| ∧ \| << \| >> \| == \| != \| < \| > \| <= \| >= |
| *For loop op.* | *opr* | ::= | <= \| >= |
| *Expression* | $e$ | ::= | $c \mid x \mid f^{(1)}\ e \mid e\ f^{[2]}\ e \mid x\ (e^*) \mid x[e] \mid x[e][e] \mid e\ ?\ e : e$ |
| *Incr.expression* | $i$ | ::= | $x$++ \| $x$-- |
| *Statement* | $s$ | ::= | $e$ \| return $e$ \| \| $\{d^*; s^*\} \mid x{=}e \mid x[e]{=}e \mid x[e][e]{=}e$ \| if $(e)$ $s$ else $s$ \| while $(e)$ $s$ \| for$(x = e; x\ opr\ e; i)$ $s$ \| switch $(e)$ $\{w^*$default: s$\}$ |
| *Switch branch* | $w$ | ::= | case $c$: $s$ break; |
| *Fun. decl.* | $g$ | ::= | $t\ x\ (a^*)\{d^*; s^*\}$ |
| *Program* | $p$ | ::= | $d^*; g^*$ |

**Fig. 1** Grammar for the C target

3. Two-dimensional arrays are represented by an array of pointers (e.g., int *x[]) instead of standard two-dimensional arrays.
4. For-loops are restricted to the most common case where the initializer is a single assignment to a variable, and the mutator expression is simply increment or decrement by one. This covers the most commonly used C idiom, and matches the OCaml `for` loops.
5. No do-while loop commands. OCaml does not have control flow statements that correspond naturally to a do-while loop.
6. Return statements are not permitted inside switch and for statements. A return can only appear at the end of a block in a function declaration or in a terminal positions at both branches of the if statement. This restriction is enforced by the type system (See Appendix 1).

While this subset is syntactically restricted compared to full C, it still provides a semantically expressive first-order language. Many missing C constructs can be effectively simulated by the ones included: for example, arithmetical mutation operators (e.g., +=) can be simulated by assignments. Similarly do-while loops can be simulated with existing looping constructs. Since staging in MetaOCaml gives the programmer complete control over what code is generated, this imposes little burden on the programmer. This subset is particularly well-supported by many industrial-strength compilers. Giving the programmer safe access to such compilers is the main purpose of implicitly heterogeneous MSP.

$$
\begin{array}{lll}
\textit{Constant} & \hat{c} \in & \textit{Int} \cup \textit{Bool} \cup \textit{Float} \cup \textit{Char} \\
\textit{Variable} & \hat{x} \in & X \\
\textit{Unary op.} & \hat{f}^{(1)} \in & \{\texttt{cos}, \texttt{sin}, \texttt{sqrt}, \texttt{float\_of\_int}, \texttt{int\_of\_float}\} \\
\textit{Limit op.} & \hat{f}^{(2)} \in & \{\texttt{min}, \texttt{max}\} \\
\textit{Binary op.} & \hat{f}^{[2]} \in & \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{+.}, \texttt{-.}, \texttt{*.}, \texttt{/.}, \texttt{**}, \texttt{mod}, \texttt{land}, \texttt{lor}, \texttt{lxor}, \\
 & & \texttt{lsl}, \texttt{lsr}, \texttt{asr}, \texttt{=}, \texttt{<>}, \texttt{<}, \texttt{>}, \leq, \geq, \texttt{\&\&}, \texttt{||}\} \\
\textit{Expression} & \hat{e} ::= & \hat{c} \mid \hat{x} \mid \hat{x}\ (\hat{e}^*) \mid \hat{f}^{(1)}\ \hat{e} \mid \hat{f}^{(2)}\ \hat{e}\ \hat{e} \mid \hat{e}\ \hat{f}^{[2]}\ \hat{e} \mid \texttt{ if } \hat{e} \texttt{ then } \hat{e} \texttt{ else } \hat{e} \\
 & & \mid \texttt{!}\hat{x} \mid \hat{x}\texttt{.(}\hat{e}\texttt{)} \mid \hat{x}\texttt{.(}\hat{e}\texttt{).(}\hat{e}\texttt{)} \\
\textit{Statement} & \hat{d} ::= & \hat{e} \mid \hat{d}; \hat{d} \mid \texttt{let } \hat{x} \texttt{ = } \hat{e} \texttt{ in } \hat{d} \mid \texttt{let } \hat{x} \texttt{ = ref } \hat{c} \texttt{ in } \hat{d} \\
 & & \mid \texttt{let } \hat{x} \texttt{ = Array.make } \hat{c}\ \hat{c} \texttt{ in } \hat{d} \mid \hat{x}\texttt{.(}\hat{e}\texttt{)} \leftarrow \hat{e} \\
 & & \mid \texttt{let } \hat{x} \texttt{ = Array.make\_matrix } \hat{c}\ \hat{c}\ \hat{c} \texttt{ in } \hat{d} \mid \hat{x}\texttt{.(}\hat{e}\texttt{).(}\hat{e}\texttt{)} \leftarrow \hat{e} \\
 & & \mid \hat{x}\texttt{:=}\ \hat{e} \mid \texttt{if } \hat{e} \texttt{ then } \hat{d} \texttt{ else } \hat{d} \mid \texttt{while } \hat{e} \texttt{ do } \hat{d} \texttt{ done} \\
 & & \mid \texttt{for } \hat{x}\texttt{=}\hat{e} \texttt{ to } \hat{e} \texttt{ do } \hat{d} \texttt{ done} \mid \texttt{for } \hat{x}\texttt{=}\hat{e} \texttt{ downto } \hat{e} \texttt{ do } \hat{d} \texttt{ done} \\
 & & \mid \texttt{match } \hat{e} \texttt{ with } ((\hat{c} \rightarrow \hat{d})^* \mid \_ \rightarrow \hat{d}) \\
\textit{Program} & \hat{s} ::= & \lambda(x^*).(\hat{d} : \hat{b}) \mid \texttt{let } \hat{x} = \hat{c} \texttt{ in } \hat{s} \mid \texttt{let } f(\hat{x}^*) = \hat{d} \texttt{ in } \hat{s} \\
 & & \mid \texttt{let } \hat{x} \texttt{ = ref } \hat{c} \texttt{ in } \hat{s} \mid \texttt{let } \hat{x} \texttt{ = Array.make } \hat{c}\ \hat{c} \texttt{ in } \hat{s} \\
 & & \mid \texttt{let } \hat{x} \texttt{ = Array.make\_matrix } \hat{c}\ \hat{c}\ \hat{c} \texttt{ in } \hat{s}
\end{array}
$$

**Fig. 2** OCaml subset grammar

## 3.3 Types in Base Language

We now turn to the base language representation of the target language subset: the types in the OCaml subset must match those of the C subset. OCaml base types `int`, `bool`, `char`, and `float` map to C int, char and double[*2], (OCaml booleans are simply mapped to C integers). The OCaml reference type (`ref`) is used to model C variables of simple type. One- and two-dimensional arrays are represented by OCaml $\hat{b}$ `array` and $\hat{b}$ `array array` types. To reflect the different restrictions on them, we will also distinguish types for function arguments, variable types and function declarations. The resulting types are as follows:

$$
\begin{array}{lllll}
\textit{Base} & \hat{b} & \in & \{\texttt{int}, \texttt{bool}, \texttt{char}, \texttt{float}\} \\
\textit{Reference} & \hat{r} & ::= & \hat{b}\ \texttt{ref} \\
\textit{Array} & \hat{a} & ::= & \hat{b}\ \texttt{array} \mid \hat{b}\ \texttt{array array} \\
\textit{Argument} & \hat{p} & ::= & \hat{b} \mid \hat{a} \\
\textit{Variables} & \hat{t} & ::= & \hat{b} \mid \hat{r} \mid \hat{a} \\
\textit{Function} & \hat{u} & ::= & (\hat{p}_0, \ldots, \hat{p}_n) \rightarrow \hat{b}
\end{array}
$$

## 3.4 Syntax for Base Language

The syntax of the source subset is presented in Figure 2. For readability,

[*2] The OCaml type `float` corresponds semantically to the C type double.

meta-variables that range over the source (OCaml) syntax are decorated with hats (e.g., $\hat{e}$) to distinguish them from meta-variables ranging over target (C) syntax. Semantically, the syntactic subset defined in this figure represents a first-order subset of OCaml with arrays and reference cells. Syntactically, `let` bindings are used to represent C declarations. Bindings representing C function declarations are further restricted to exclude function declarations nested in the bodies of functions. Assignments in C are represented by updating OCaml references or arrays. The grammar for a *Program* ensures there is exactly one top level entry function. This top level function will be mapped to main in the generated C program.

## 3.5 Offshoring Translation

This section formalizes the translation from the source subset to C. OCaml types and expressions are translated directly into C types and expressions.

$\boxed{[\![\hat{t}]\!]}$

$$
\begin{array}{rcl}
[\![\texttt{int}]\!] & = & \mathsf{int} \\
[\![\texttt{bool}]\!] & = & \mathsf{int} \\
[\![\texttt{char}]\!] & = & \mathsf{char} \\
[\![\texttt{float}]\!] & = & \mathsf{double} \\
[\![\hat{b}\ \texttt{array}]\!] & = & [\![\hat{b}]\!]\ [\,] \\
[\![\hat{b}\ \texttt{array array}]\!] & = & *[\![\hat{b}]\!][\,]
\end{array}
$$

$\boxed{[\![\hat{e}]\!]}$

$$
\begin{array}{rcl}
[\![\hat{c}]\!] & = & c \\
[\![\hat{x}]\!] & = & x \\
[\![\hat{x}(\hat{e_1}, \ldots, \hat{e_n})]\!] & = & x([\![\hat{e_1}]\!], \ldots, [\![\hat{e_n}]\!]) \\
[\![\hat{f}^{(1)}\hat{e}]\!] & = & [\![\hat{f}^{(1)}]\!][\![\hat{e}]\!] \\
[\![\hat{f}^{(2)}\hat{e_1}\hat{e_2}]\!] & = & [\![\hat{f}^{(2)}]\!][\![\hat{e_1}]\!][\![\hat{e_2}]\!] \\
[\![\hat{e_1}\hat{f}^{[2]}\hat{e_2}]\!] & = & [\![\hat{e_1}]\!][\![\hat{f}^{[2]}]\!][\![\hat{e_2}]\!] \\
[\![\texttt{if}\ \hat{e_1}\ \texttt{then}\ \hat{e_2}\ \texttt{else}\ \hat{e_3}]\!] & = & [\![\hat{e_1}]\!]?[\![\hat{e_2}]\!] : [\![\hat{e_3}]\!] \\
[\![!\hat{x}]\!] & = & x \\
[\![\hat{x}.(\hat{e})]\!] & = & x[[\![\hat{e}]\!]] \\
[\![\hat{x}.(\hat{e_1}).(\hat{e_2})]\!] & = & x[[\![\hat{e_1}]\!]][[\![\hat{e_2}]\!]]
\end{array}
$$

Recall that we distinguish meta-variables ranging over OCaml- and C-constructs using the hat notation $x$ vs. $\hat{x}$. Moreover, the translations from OCaml to C subset that map variable names in the source language to the (same) variable names in the target language are written by erasing the hat notation from the meta-variable: $[\![\hat{x}]\!] = x$. This meta-level operation of "erasing the hat" is in fact an identity, since both syntactic subsets share the same set of variable names $X$, i.e., for any $x \in X$, we have $\hat{x}$ and $x$ refer to the same (object-level) variable. A similar convention is employed for constants as well throughout our formal development.

The statement subset of OCaml is translated to a pair $(\overline{l}, \overline{s})$, where $\overline{l}$ are declarations of variables that are bound in OCaml `let` expressions, and $\overline{s}$, the sequence of $C$ statements that corresponds to OCaml statements. The translation for OCaml statements $\{\!|\hat{d}|\!\}$ is written with a *return context* which can be $\bot$ or $\top$.

$$\textit{Return context} \quad a \quad ::= \quad \bot \mid \top$$

If the return context is $\bot$, the translation does not generate return statements at the leaves; on the other hand, if the return context is $\top$, bottoming out at a leaf OCaml expression produces the appropriate C return statement.

$\boxed{\{\!|\hat{d}|\!\}_a = (l, s)}$

$$\frac{}{\{\!|\hat{e}|\!\}_\bot = (\cdot, [\![\hat{e}]\!])} \qquad \frac{}{\{\!|\hat{e}|\!\}_\top = (\cdot, \texttt{return}\ [\![\hat{e}]\!])} \qquad \frac{}{\{\!|\hat{x} := \hat{e}|\!\}_\bot = (\cdot, x = [\![\hat{e}]\!])} \qquad \frac{}{\begin{array}{l}\{\!|\quad \hat{x}.(\hat{e_1}) \leftarrow \hat{e_2}|\!\}_\bot = \\ (\cdot, x[[\![\hat{e_1}]\!]] = [\![\hat{e_2}]\!])\end{array}}$$

$$\frac{}{\begin{array}{l}\{\!|\hat{x}.(\hat{e_1}).(\hat{e_2}) \quad \leftarrow \hat{e_3}|\!\}_\bot = \\ (\cdot, x[[\![\hat{e_1}]\!]][[\![\hat{e_2}]\!]] = [\![\hat{e_3}]\!])\end{array}} \qquad \frac{\{\!|\hat{d}|\!\}_a = (l, s)}{\{\!|\left\{\begin{array}{l}\texttt{let}\ \hat{x} : \hat{t} = \hat{e} \\ \texttt{in}\ \hat{d}\end{array}\right\}|\!\}_a = ((([\![\hat{t}]\!] x; l), (x = [\![\hat{e}]\!]; s))}$$

$$\frac{\begin{array}{c}\{\!|\hat{d_1}|\!\}_\bot = (l_1, s_1) \\ \{\!|\hat{d_2}|\!\}_a = (l_2, s_2)\end{array}}{\{\!|\hat{d_1}; \hat{d_2}|\!\}_a = (l_1; l_2, s_1; s_2)} \qquad \frac{\{\!|\hat{d}|\!\}_a = (l, s)}{\{\!|\texttt{let}\ \hat{x} : \texttt{ref}\ \hat{t} = \texttt{ref}\ \hat{c}\ \texttt{in}\ \hat{d}|\!\}_a = (([\![t]\!]\, x; l), (x = c; s))}$$

$$\frac{\{\!|\hat{d}|\!\}_a = (l, s)}{\{\!|\left\{\begin{array}{l}\texttt{let}x : \hat{t}\,\texttt{array} = \\ \texttt{Array.make}\ \hat{c_1}\ \hat{c_2}\ \texttt{in}\ \hat{d}\end{array}\right\}|\!\}_a = \begin{array}{l}([\![t]\!]\ x[\,] = \{\overline{c_2}^{\,c_1\ times}\}, l\,;\ s)\end{array}} \qquad \frac{\{\!|\hat{d}|\!\}_a = (l, s) \quad \overline{y_m}\ fresh}{\begin{array}{l}\{\!|\texttt{let}\ \hat{x} : \hat{t}\texttt{array array} = \\ \texttt{Array.make\_matrix}\ \hat{e}\ \hat{c_2}\ \hat{c_3}\ \texttt{in}\ \hat{d}|\!\}_a = \\ (\overline{[\![\hat{t}]\!]\ y_m = \{\overline{[\![e]\!]}^{\,c_2\ times}\}}^{\,c_1\ times}\ ; \\ [\![\hat{t}]\!]\ *x[\,] = \{y_1, \ldots, y_{c_1}\}; l, s)\end{array}}$$

$$\frac{\begin{array}{c}\{\!|\hat{d_1}|\!\}_a = (l_1, s_1) \\ \{\!|\hat{d_2}|\!\}_a = (l_2, s_2)\end{array}}{\begin{array}{l}\{\!|\texttt{if}\ (\hat{e})\ \texttt{then}\ \hat{d_1}\ \texttt{else}\ \hat{d_2}|\!\}_a = \\ (\cdot, \mathsf{if}([\![\hat{e}]\!])\{l_1; s_1\}\ \mathsf{else}\ \{l_2; s_2\})\end{array}} \qquad \frac{\{\!|\hat{d}|\!\}_\bot = (l, s)}{\begin{array}{l}\{\!|\texttt{while}\ (\hat{e})\ \texttt{do}\ \hat{d}\ \texttt{done}|\!\}_\bot = \\ (\cdot, \mathsf{while}([\![\hat{e}]\!])\{l; s\})\end{array}}$$

$$\frac{\{\!|\hat{d}|\!\}_\bot = (l, s)}{\begin{array}{l}\{\!|\texttt{for}\ \hat{x} = \hat{e_1}\ \texttt{to}\ \hat{e_2}\ \texttt{do}\ \hat{d}\ \texttt{done}|\!\}_\bot = \\ (\mathsf{int}\ x, \mathsf{for}\ (x = [\![\hat{e_1}]\!]; x\texttt{<=}[\![\hat{e_2}]\!]; x\texttt{++})\{l; s\})\end{array}} \qquad \frac{\{\!|\hat{d}|\!\}_\bot = (l, s)}{\begin{array}{l}\{\!|\texttt{for}\ \hat{x} = \hat{e_1}\ \texttt{downto}\ \hat{e_2}\ \texttt{do}\ \hat{d}\ \texttt{done}|\!\}_\bot \\ = (\mathsf{int}\ x, \mathsf{for}\ (x = [\![\hat{e_1}]\!]; x\texttt{>=}[\![\hat{e_2}]\!]; x\texttt{--})\{l; s\})\end{array}}$$

$$\frac{\{\!|\hat{d_n}|\!\}_{\bot\, n} = (l_n, s_n) \quad \{\!|\hat{d}|\!\}_\bot = (l, s)}{\{\!|\texttt{match}\ \hat{e}\ \texttt{with}\ \overline{\hat{c_n}\ \rightarrow\ \hat{d}_{n\, n}}\ |\ \_ \rightarrow\ \hat{d}|\!\}_\bot = \begin{array}{l}\mathsf{switch}([\![\hat{e}]\!])\{ \\ \overline{\mathsf{case}\ c_n : \{l_n; s_n; \mathsf{break}\};} \\ \mathsf{default} : \{l; s\}\}\end{array}}$$

OCaml programs $\hat{s}$ are translated into a pair $(g, l)$, where $g$ is a sequence of function definitions and $l$ is a sequence of variable declarations. Note, we use the name procedure for our main function, which we assume to be different from the name of any other function to avoid name clashes with other procedure names.

$\boxed{\langle\!|\hat{s}|\!\rangle = (g, l)}$

$$\frac{\langle\!|\hat{s}|\!\rangle = (g,l)}{\langle\!|\texttt{let } \hat{x} : \hat{t} = \hat{c} \texttt{ in } \hat{s}|\!\rangle = (g, ([\![\hat{t}]\!]\ x\ =\ c; l))}$$

$$\frac{\langle\!|\hat{s}|\!\rangle = (g,l)}{\langle\!|\texttt{let } \hat{x} : \texttt{ref } \hat{t} = \texttt{ref } \hat{c} \texttt{ in } \hat{s}|\!\rangle = (g,\ [\![\hat{t}]\!]x = c; l)}$$

$$\frac{\langle\!|\hat{s}|\!\rangle = (g,l)}{\langle\!|\left\{\begin{array}{l}\texttt{let } \hat{x} : \hat{t} \texttt{ array} = \\ \texttt{Array.make } \hat{c_1}\ \hat{c_2} \texttt{ in } \hat{s}\end{array}\right\}|\!\rangle = (g,\ [\![\hat{t}]\!]\ x[\,] = \{\overline{c_2}^{c_1\ times}\}; l)}$$

$$\frac{\langle\!|\hat{d}|\!\rangle = (g,l) \quad (y_i\ fresh)^{i\in\{1\ldots c_1\}}}{\langle\!|\texttt{let } \hat{x} : \hat{t} \texttt{ array array} = \texttt{Array.make\_matrix } \hat{e}\ \hat{c_2}\ \hat{c_3} \texttt{ in } \hat{d}|\!\rangle_a = (g, \left([\![\hat{t}]\!]\ y_i = \{\overline{[\![e]\!]}^{c_2\ times}\}\right)^{i\in\{\ldots c1\}}; [\![\hat{t}]\!]\ *x[\,] = \{y_1, \ldots, y_{c_1}\}; l)}$$

$$\frac{\langle\!|\hat{s}|\!\rangle = (g,l) \quad \{\!|\hat{d}|\!\}_\top = (l_d, s_d)}{\langle\!|\texttt{let } \hat{f}\ (\hat{x_i} : \hat{p_i})^{i\in\{\ldots n\}} : \hat{b} = \hat{d} \texttt{ in } \hat{s}|\!\rangle = ([\![\hat{b}]\!] f\ ([\![p]\!]_i\ x_i)^{i\in\{\ldots n\}}\ \{l_d; s_d;\}; g, l)}$$

$$\frac{\{\!|\hat{d}|\!\}_\top = (l,s)}{\langle\!|\lambda\ (\hat{x_i} : \hat{p_i})^{i\in\{\ldots n\}}.(\hat{d} : \hat{b})|\!\rangle = (([\![\hat{b}]\!]\ \textsf{procedure}\ ([\![p_i]\!]\ x_i)^{i\in\{\ldots n\}}\ \{l; s\}), \cdot)}$$

## 3.6 What the Programmer Sees

The programmer is given access to offshoring simply by making MetaOCaml run construct (`.!`$e$) customizable. In addition to this standard form, the programmer can now write `.!{Trx.run_gcc}`$e$ or `.!{Trx.run_icc}`$e$ to indicate that offshoring should be used, and that either the `gcc` or `icc` compilers, respectively, should be used to compile the resulting code. Additional arguments can also be passed to these constructs. For example, the programmer can write `{Trx.run_gcc}`$e$ (resp. `{Trx.run_icc}`$e$) as follows:

```
.!{Trx.run_gcc with compiler = "cc"; compiler_flags="-g -O2"} ...
```

to use an alternative compiler `cc` with the flags `-g -O2`.

**Programming with offshoring** We illustrate offshoring in MetaOCaml with a small example and comment on the specific issues that arise when writing program generators that produce offshorable code. Consider the following generator that produces the body of a exponentiation function, specialized on the exponent, by repeatedly multiplying an accumulator variable by the base:

```
(* bodyGen: int -> int code -> (int ref) code -> int code *)
let rec bodyGen n x accum =
  if n = 0
  then .< ! .~(accum) >.
  else .<  (.~(accum) := .~x * ! .~(accum); .~(bodyGen (n-1) x accum)) >.
```

If `n` is 3, `bodyGen` generates 3 updates to the accumulator and returns the accumulator. Using `bodyGen`, we can write a function `powerGen` which generates a specialized exponentiation function, and passes it as an argument to another

program generator `f`, which produces another piece of code which may splice in the specialized power function at function call sites:

```
(* powerGen : int -> ((int -> int) code -> 'a code) -> 'a code *)
let powerGen n f =
  .< let pwr x =
       let accum = ref 1 in .~(bodyGen n .<x>. .<accum>.)
     in .~(f .<pwr>. ) >.
```

If the programmer wants to offshore the results of `powerGen`, he must ensure that any use of `powerGen` respects the following invariants: (1) `powerGen` should only be spliced into a context representing top-level C programs (i.e., wherever the nonterminal $\hat{s}$ in Figure 2 can be used); (2) any "consumer" code generator `f` given to `powerGen` should only produce a top-level C program; (3) `f`'s argument should only be spliced into a function application context ( $\hat{x}$ in the production $\hat{e} ::= \hat{x}\ (\hat{e}^*)$ of Figure 2). Understanding and maintaining such invariants is the programmer's responsibility, but it is relatively easy in practice because MSP gives him complete control over the structure of the generated code.

Below, we show a correct use of `powerGen` (the corresponding C code produced by offshoring translation is shown to the right of code generated by MetaOCaml):

```
let program = powerGen 3 (fun cube -> .< fun x ->  (.~cube x)-1>. )
let result  = .!{Trx.run_gcc} program

val program : ('a, int -> int) code =
.<let pwr3 x  =              (*      int pwr3 (int x)              *)
                             (*       {                            *)
   let accum  = ref 1 in     (*         int accum;                 *)
                             (*         accum = 1;                 *)
   accum := x * ! accum;     (*         accum = x * accum;         *)
   accum := x * ! accum;     (*         accum = x * accum;         *)
   accum := x * ! accum;     (*         accum = x * accum;         *)
   ! accum in                (*         return accum;              *)
                             (*       }                            *)
                             (*       int procedure_main (int x) { *)
                             (*       {                            *)
  fun x ->  (pwr3 x)-1>.     (*         return (pwr3(x)-1);        *)
                             (*       }                            *)
val result : int -> int = <fun>
```

In the following example, however, the invariant (1) is violated, because `powerGen` is used inside a function body definition, and results in code that cannot be offshored (evaluating `(.!{Trx.run_gcc} wrong)` raises an exception):

```
let wrong = .< fun x -> .~(powerGen 2 (fun square -> .< .~square (x) >.)) >.
```

```
val wrong : ('a, int -> int) code =
  .<fun x ->
   let pwr_2 x  =
    let accum  = (ref 1) in
    accum := x * ! accum;
    accum := x * ! accum;
    ! accum in
   (pwr_2 x)>.
```

### 3.7 Marshalling and Dynamic Linking

The C and OCaml runtime systems use different representations for values. Therefore, inputs and outputs of offshored functions must be marshalled from one representation to the other.

The offshoring implementation automatically generates a marshalling wrapper C function for the top-level entry point of the offshored function. The marshalling function depends only on the type of the top-level function. First, the OCaml runtime values are converted to their C representations. The standard OCaml library provides conversions for base types. We convert arrays by allocating a new C array, converting each element of the OCaml array, and storing it in the C array. The C function is invoked with the marshalled C values. Its results are collected, and marshalled back into OCaml. To account for updates in arrays, the elements of the C are converted and copied back into the OCaml array.

Once a C program and the marshalling code has been produced, the result is then compiled into a shared library (a `.so`) file. To load this shared library into MetaOCaml's runtime system, we extend Stolpmann's dynamic loading library for OCaml [14] to support dynamic loading of function values.

## §4 Type Preservation

Showing that an offshoring translation can preserve typing requires formalizing the type system for the target language, the source language, as well as the translation itself.

The type system for top-level statements in OCaml programs is defined by the derivability of the judgment $\hat{\Gamma} \vdash \hat{p} : \hat{t}$ (Appendix 2). Similarly, the type system for C programs is defined by a judgment $\Gamma \vdash g$ (Appendix 1). Section 3.5 provided the definition of translation functions. For example, $\langle\!|\hat{s}|\!\rangle = (l_1, \ldots, l_m, g_1, \ldots, g_n)$ translates a top-level OCaml program

into a set of C variable declarations and function definitions, and $[\![\hat{\Gamma}]\!]$ translates the OCaml variable and function environment, $\hat{\Gamma}$, into the corresponding C environment. We define an operation $|\cdot|$ on variable declarations and function definitions that translates them into a C type environment (Appendix 4).

Any valid term in the base language translates to a valid one in the target language.

**Theorem 4.1 (Type Preservation)**
If $\hat{\Gamma} \vdash \hat{s} : \hat{a_n} \rightarrow \hat{b}$ and $\langle\!|\hat{s}|\!\rangle = (g, l)$, then $[\![\hat{\Gamma}]\!] \cup |l| \cup |g| \vdash g$.

**Proof** By induction over the height of first derivation. The details of the proof are presented in Appendix 4. ■

Proving this theorem is work that is done only once per offshoring translation, and is done by the language designer. Users of offshoring are relieved from the difficulty of ensuring that their program generators produce well-typed target-language (C) programs. The programmer only needs to establish, on a per-generator basis, the relatively easier invariant that a particular program generator produces code that is in the offshorable syntactic subset.

# §5 Effect on Performance Characteristics

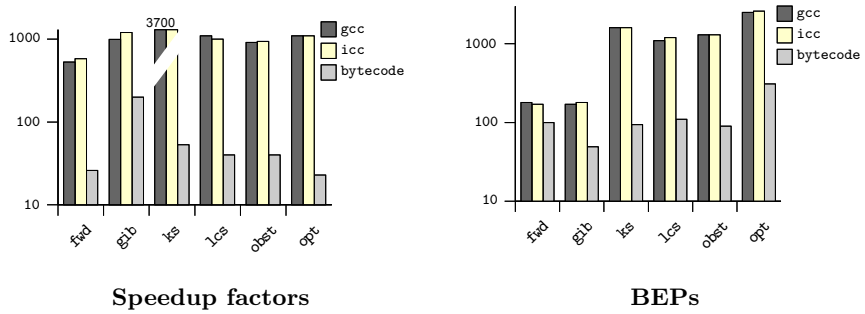
This section summarizes the empirical measurements gathered to evaluate the performance impact of offshoring. The questions we want these experiments to answer are: How does the performance of offshored code compare to that of the same code executed with *run*? Does marshalling impose a significant overhead on offshoring? As MetaOCaml currently only supports bytecode OCaml execution, would extending MetaOCaml to support native OCaml-style compilation be an acceptable alternative to offshoring?

## 5.1 Benchmarks

As a benchmark, we use a suite of staged dynamic programming algorithms. These algorithms were initially implemented to study staging of dynamic programming algorithms [15]. The benchmark consists of (both unstaged and staged) MetaOCaml implementations of the following algorithms [2]:

- `forward`, the forward algorithm for Hidden Markov Models. Specialization size (size of the observation sequence) is 7.
- `gib`, the Gibonacci function, a minor generalization of the Fibonacci sequence. Specialization is for $n = 25$.

**Fig. 3** Speedups and break-even points




- `ks`, the 0/1 knapsack problem. Specialization is for size 32 (number of items).
- `lcs`, the least common subsequence problems. Specialization is for string sizes 25 and 34 for the first and second arguments, respectively.
- `obst`, the optimal binary search tree algorithm. Specialization is a leaf-node-tree of size 15.
- `opt`, the optimal matrix multiplication problem. Specialization is for 18 matrices.

To evaluate offshoring, we compare executing the result of these algorithms in MetaOCaml to executing the result in C using offshoring. Appendix 5 documents the measurements in tabular from. In this section, graphs will be used to summarize the results. Timings were collected on a Pentium 4 machine (3055MHz, 8K L1 and 512K L2 cache, 1GB main memory) running Linux `2.4.20-31.9`. All experiments are fully automated and available online [3]. We report results based on version `DP_002/1.25` of the benchmark. It was executed using MetaOCaml version `308_alpha_020` bytecode compiler, Objective Caml version `3.08.0` native code compiler, and GNU's gcc version `2.95.3 20010315`, and Intel's icc version `8.0 Build 20031016Z` C compilers.

## 5.2 Offshoring vs. OCaml Byte Code Compiler

Because of engineering issues with OCaml's support for dynamic loading, MetaOCaml currently extends only the bytecode compiler with staging constructs, but not the native code compiler. We therefore begin by considering the impact of offshoring in the bytecode setting.

Each of the benchmark programs comes in two versions: unstaged and staged. A staged version is obtained by turning the unstaged version into a program generator by specializing it with respect to a subset of its arguments. The resulting program computes the same result as the unstaged version, but in two stages: the first, where a program is generated and compiled; the second, where the generated program is executed with the remaining arguments. The ratio between the execution time of the unstaged version and the *second* stage of the staged version (with the same inputs and producing the same outputs) is reported as the *speedup*. A *break-even point* (BEP) is the number of times that the second stage must be executed until the initial overhead of generating and compiling it (i.e., the first stage) pays off. For example, although the speedup gained by staging the `forward` algorithm and compiling it with `gcc` is 530 times, we need at least 180 uses of it before is cheaper to use the staged version than to just use the unstaged one.

The left-hand side of the Figure 3 summarizes the comparative speedups gained by staging the benchmark algorithms with (`gcc` and `icc`) and without offshoring (`bytecode`). Performance of offshoring is measured using two different C compilers (the GNU `gcc` and Intel's `icc`). The right-hand side of the Figure 3 compares the break-even points for the same executions.

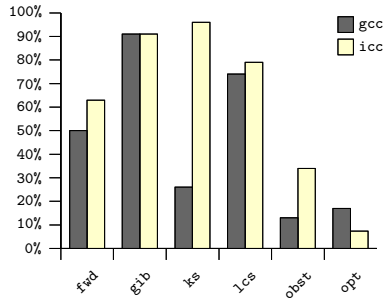
In general, the results obtained for offshoring to C are encouraging. The speedups from offshoring and compiling with the C compilers are always greater than 1, and are often an order of magnitude greater. But it should be noted that the BEP's are higher for offshored than for the non-offshored staged execution within MetaOCaml. This is primarily a result of the C compilation times being higher than OCaml bytecode compilation times.

## 5.3 Marshalling overhead

To assess the impact of marshalling on the runtime performance, we compare the time to execute the benchmarks from within MetaOCaml against the time it took to execute the same functions in a C program with no marshalling. Each benchmark program is compiled by the `gcc` and `icc` compilers. Figure 4 displays the percentage of the total execution time for each benchmark spent in marshalling (`gcc` or `icc`).

In the `forward` example, we marshall a 7-element integer array. In `gib`, we marshall only two integers, and thus have a low marshalling overhead in absolute terms. But because the total computation time in this benchmark

**Fig. 4** Marshalling overhead for `gcc` and Intel's `icc` compiled offshored code




is small, the marshalling overhead percentage is high. In `ks`, we marshall a 32-element integer array, and in `lcs` benchmark, we marshall two 25- and 34-element character arrays, which accounts for the high marshalling overhead in these benchmarks in absolute terms. Since both these benchmarks perform a significant amount of computation, the proportion of time spent marshalling may be lower than `gib` or `forward`. Similarly, `obst` and `opt` have significant marshalling overheads in absolute terms since they must marshall 15-element floating-point and 18-element integer arrays.

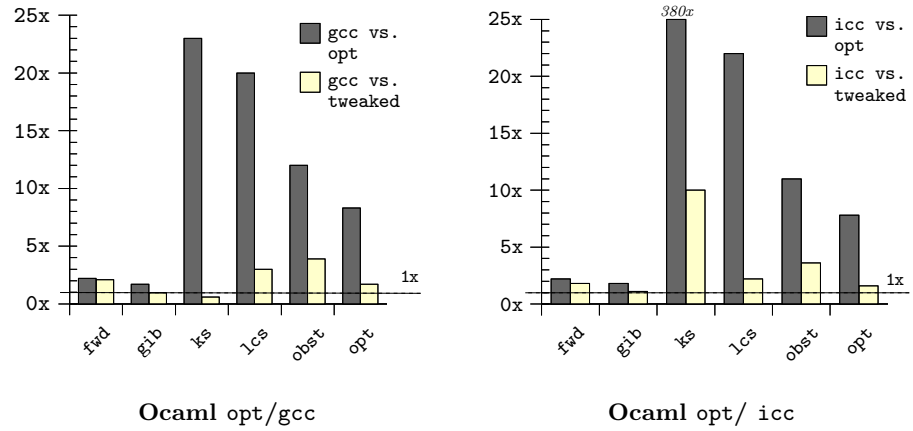
The data indicates that marshalling overhead is significant. We chose the current marshalling strategy for implementation simplicity, and not for efficiency. These results suggest that it would be useful to explore an alternative marshalling strategy that allows sharing of the same data-structures across the OCaml/C boundary.

## 5.4 Offshoring vs. OCaml Native Compiler

One motivation for offshoring is that standard implementations of high-level languages are unlikely to compile automatically generated programs as effectively as implementations for lower-level languages such as C. The figures reported so far relate to the MetaOCaml bytecode compiler. How does offshoring perform against the native code compiler?

Figure 5 reports the speedup factor achieved by offshored programs over the same programs compiled using the native-code OCaml compiler (known as `opt`), where speedup factor greater than one mean that the offshored version is faster. Two kinds of comparisons are made: first, where stage two programs are compiled using the native OCaml compiler (labeled `opt vs. gcc` and `opt vs.`

**Fig. 5** The speedup of offshored vs. OCaml optimizing native compiler code




`icc`); and second, where stage two programs are first hand-modified ("tweaked") before they are compiled by the native OCaml compiler (labeled `tweaked vs. gcc` and `tweaked vs. icc`). The tweaking modifications involve applying the following simple transformations to the code generated by MetaOCaml: currying the arguments to functions, replacing the uses of the polymorphic OCaml operators `min` and `max` by their monomorphic versions, and moving array initializations outside the generated functions.[*3]

Tweaking has been used to obtain a fairer performance comparison by addressing some issues that arise only with the OCaml native compiler. For example, the benchmark `ks` contains a large number of calls to the `max` function. The default OCaml implementation of `max` is polymorphic to allow it to work on all numeric types, and involves a runtime check to determine the type of its arguments.[*4] Replacing the polymorphic `max` with a monomorphic version in the tweaked code can easily account for vast speedups of C over OCaml in the untweaked case, and a significant performance benefit from tweaking.

Without tweaking, offshoring always outperforms the native code compiler. After tweaking the generated code, the situation is slightly different. In

[*3] Thanks to Xavier Leroy for pointing out that these changes would improve the relative performance of the OCaml native code compiler.

[*4] The code generated by the native code compiler for the library function `max` takes about 15 assembly instructions on the x86 architecture, including a costly indirect function call to a C function in the runtime library. On the other hand, a simple monomorphic version of `max` is compiled down to 4 very efficient register-based instructions.

two cases, the OCaml native code compiler outperforms `gcc`. In fact, for the `ks` benchmark, the OCaml compiler does significantly better than `gcc` (by a factor of 2).

The tweaks that improved the relative performance of OCaml's native code compiler draw attention to an important issue in designing an offshoring transformation and in interpreting speedups such as the ones presented here: Speedups are sensitive to the particular representation that we chose in the base language, and the real goal is to give the programmer the ability to generate specific programs in the target language. Speedups reported here give one set of data-points exhibiting the potential of offshoring.

## §6 Further Opportunities for Offshoring

The formal translation presented here focuses on ensuring that the generating program is well-typed, and makes some simplifying assumptions about the semantics of OCaml and C. For example, OCaml's default integer type is 31 bits, while in C it is 32. This mismatch can be addressed by using OCaml 32 bits integers in the source language. This would, however, bias the performance numbers in favor of offshoring, as OCaml's default 31 bit integers are faster. Our intent is not that providers of offshoring translators verify their translators formally. As with any compiler, the semantic fidelity of the offshoring translation can depend on the application, and would be established in a manner that is no different than that for a compiler.

The particular offshoring translation presented here is useful for building multi-stage implementations of several standard algorithms. But there are other features of C that have distinct performance characteristics, and which the programmer might wish to gain access to. We expect that the translation presented here can be expanded systematically: An offshoring translation can be viewed as an inverse of a total map from the target to the source language. The backwards map would be a kind of OCaml semantics for the constructs in the target language. With this insight, several features of C can be represented in OCaml as follows: Function pointers can be handled naturally in a higher order language, although the representative base language subset would have to be restricted to disallow occurrences of free variables in nested functions. Such free variables give rise to the need for closures when compiling functional languages, and our goal is not to compile, but rather to give the programmer access to the notion of function pointers in C. If the programmer wishes to implement closures

in C, it can be done explicitly at the OCaml level. For control operators, we can use continuations, and more generally, a monadic style in the source language, but the source would have similar restrictions on free variables to ensure that no continuation requires closures to be (automatically) constructed in the target code. Targeting struct and union types should be relatively straightforward using OCaml's notion of datatypes. Dynamic memory management, bit manipulation, and pointer arithmetic can be supported by specially marked OCaml libraries that simulate operations on an explicit memory model in OCaml.

OCaml Bigarrays can be used instead of arrays of pointers for 2- dimensional arrays. Bigarrays allow sharing of OCaml arrays with C and would be useful in a number of ways: they support more C types, they may be multi-dimensional, they don't need marshalling, and they don't require initialization.

The offshoring strategy proposed here relies on the programmer to check the invariant that a particular program generator produces code that is in the offshorable syntactic subset. In principle, static analysis could be used to check this invariant. It is likely, however, that this would be of the same complexity as inferring refinement types. This remains as an interesting question for future work.

# §7 Related Work

**Runtime Code Generation** At first sight, implicitly heterogeneous MSP seems closely related to runtime-code generation. However, implicitly heterogenous MSP has different goals from runtime low-level code generation (RTCG) [26, 27, 28, 29]. In particular, with RTCG the goal is to compile all code (even second-stage code) into machine code at the compilation time of the generator. Compiling at this time means that compilers can only work on small code fragments, and there are generally less opportunities for optimization than if compilation happens after all code is generated and combined. RTCG is relevant only if very fast generation times are desirable, even at the cost of reduced performance for the generated code. Our work assumes a setting where the primary concern is the runtime performance of the generated code, not the generator.

**Heterogeneous Quasi-quotation** Kamin [8, 9] describes an approach to implementing domain specific languages by defining a well-chosen set of combinators in a functional language (Standard ML). These combinators are designed

explicitly with a target language in mind. If we wanted to target a different language or, sometimes, if we wish to implement a different application, the process of design and implementation of combinators must be repeated from scratch.

**Typeful Metaprogramming with GADTs** Given a meta-language with a sufficiently expressive type system [13, 22, 23, 24, 25], the programmer can encode object-language abstract syntax trees that are statically guaranteed to represent only well-typed object-language programs. For example, the following Haskell program uses a generalized algebraic data-type to represent well-typed terms of the simply typed $\lambda$-calculus.

```
data Exp e t =
   Var    :: Exp (e,t) t
   Shift  :: Exp e t -> Exp (e,t1) t
   Abs    :: Exp (e,t1) t2 -> Exp e (t1 -> t2)
   App    :: Exp e (t1 -> t2) -> Exp e t1 -> Exp t2
```

The representation above relies on meta-language types to represent types in the object language. Type systems with GADTs are sufficiently expressive to statically ensure that program generators maintain typing invariants of object-language programs they manipulate. Their main advantage over the approach presented here are expressiveness and user-extensibility: the programmer need not rely on the language designer to craft offshoring translations and target multiple object-languages. However, in case where the target language is a small, well-designed subset (as the subset of C presented here), the GADT based approaches seem less practical then the approach we present in this paper.

A practical drawback of the GADT approach is that it requires that object-language binding constructs are encoded using de Bruijn indices. The correctness of such an encoding with respect to the typing rules must be established by the programmer. Offshoring automatically ensures the hygienic and type-safe manipulation of object-language variables. It also allows the use quotation mechanisms. Finally, the programmer using GADTs cannot reuse the same program generator to target multiple object languages. Rather, for each new object-language encoding the program generator must be reimplemented from scratch.

# §8 Conclusion

We have proposed the idea of implicitly heterogeneous MSP as a way of combining the benefits of the homogeneous and heterogeneous approaches. In particular, generators need not be coupled with the syntax of the target language, and the programmer need not be bound to the performance characteristics of the base language. To illustrate this approach, we target a subset of C suitable for numerical computation, and use MetaOCaml as the base language. We prove that the offshoring translation proposed for this subset preserves typing. This is done once in the language design stage, and the programmer has the same static typing guarantees of homogeneous MSP.

We have implemented this extension in MetaOCaml. Experimental results indicate that the approach can yield significantly better performance compared to the OCaml bytecode compiler, and often better performance than the OCaml native code compiler. A fully automated version of our performance measurement suite has been implemented and made available online [3)].

***Acknowledgment*** We would like to thank John Mellor-Crummey, Gregory Malecha, James Sasitorn, Jeremy Siek Dan Vanderkam, and Angela Zhu, who read drafts of this paper and gave us several helpful suggestions. A shorter version of this paper was submitted to the Generative Programming and Component Engineering (GPCE) conference. We would like to thank the GPCE and NGC reviewers and editors for their detailed and constructive comments.

## References


1) Cristiano Calcagno, Eugenio Moggi, and Walid Taha. ML-like inference for classifiers. In *European Symposium on Programming (ESOP'04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 79-93. Springer-Verlag, 2004.

2) Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 14th edition, 1994.

3) Dynamic Programming Benchmarks. Available online from `http://www.metaocaml.org/examples/dp`, 2005.

4) Jason Eckhardt, Roumen Kaiabachev, Emir Pasalic, Kedar Swadi, and Walid Taha. Implicitly heterogeneous multi-stage programming. In *Proceedings of the 4th ACM International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *Lecture Notes in Computer Science*, pages 275-292. Springer-Verlag, 2005.

5) Conal Elliott, Sigbjørn Finne, and Oege de Moore. Compiling embedded languages. In [17], pages 9–27, 2000.

6) Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

7) Robert Glück and Jesper Jørgensen. Multi-level specialization (extended abstract). In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *LNCS*, pages 326–337. Springer-Verlag, 1999.

8) Sam Kamin. Standard ML as a meta-programming language. Technical report, Univ. of Illinois Computer Science Dept, 1996. Available at `http://www-faculty.cs.uiuc.edu/~kamin/pubs/`.

9) Samuel N. Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components I: Source-level components. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 49–64, London, UK, 2000. Springer-Verlag, volume 1799 of LNCS.

10) MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2004.

11) Gregory Neverov and Paul Roe. *Towards a Fully-reflective Meta-programming Language*, volume 38 of *Conferences in Research and Practice in Information Technology*, pages 151-158. ACS, Newcastle, Australia, 2005. Twenty-Eighth Australasian Computer Science Conference (ACSC2005).

12) 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`.

13) Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *Proceedings of the International Conference on Functional Programming (ICFP '02)*, pages 218-229, Pittsburgh, USA, October 2002. ACM.

14) Gerd Stolpmann. DL- ad-hoc dynamic loading for OCaml. Available from `http://www.ocaml-programming.de/packages/documentation/dl/`.

15) Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, ACM Press, January 2006.

16) Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [12].

17) Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.

18) Walid Taha. 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 Manipulation (PEPM'00)*, pages 34-43. Boston, 2000. ACM Press.

19) Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '03)*, 2003 ACM SIGPLAN vol. 38 (num. 1), pages 26-37. New Orleans, Louisisana, 2003.

20) Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of Symposium on Partial Evaluation and Semantics Based Program manipulation*, pages 203–217. ACM SIGPLAN, 1997.

21) David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

22) Hongwei Xi and Chiyan Chen and Gang Chen. Guarded recursive datatype constructors. *In POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–235, 2003, ACM Press, New Orleans, Louisiana, USA

23) James Cheney and Ralf Hinze. First-Class Phantom Types. Technical Report 1901, Cornell University, Mar. 2006.

24) Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. *Presented at the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM'04)* Cork, Ireland, July, 2004.

25) Simon Peyton Jones and Dimitrios Vytiniotis and Stephanie Weirich and Geoffrey Washburn. Simple unification-based type inference for GADTs *In ACM SIGPLAN Notices*, vol. 41, num. 9, pages 50–61, Sep, 2006.

26) Dawson R. Engler. VCODE : A Retargetable, Extensible, Very Fast Dynamic Code Generation System *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation* pages 160–170, May, 1996, ACM Press, New York.

27) Charles Consel and François Noël A General Approach for Run-Time Specialization and its Application to C. *Conference Record of POPL '96: The*

23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pages 145–156, 21–24 Jan, 1996, St. Petersburg Beach, Florida.

28) Brian Grant and Markus Mock and Matthai Philipose and Craig Chambers and Susan J. Eggers. Annotation-Directed Run-Time Specialization in C. *In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. pages 163–178, Jun 1997, Amsterdam, The Netherlands.

29) Michael Sperber and Peter Thiemann. Two for the Price of One: Composing Partial Evaluation and Compilation *In Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*. pages 215–225, Jun, 1997, Las Vegas, Nevada.

# §1 Type System for the C Fragment

The type system for the target language is defined for a fixed type assignment $\Sigma$ that assigns types to constants and operators. We assume the type assignment $\Gamma$ must be well-formed in the sense that it assigns at most one type for each variable.

Expressions $(\Gamma \vdash e : t)$

$$\frac{c : b \ \in \ \Sigma}{\Gamma \vdash c : \ b}\texttt{Const} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t}\texttt{Var} \qquad \frac{f^{(1)} : b\ f(b_1) \in \Sigma \quad \Gamma \vdash e : b_1}{\Gamma \vdash f^{(1)}\ (e) : b}\texttt{Op1} \qquad \frac{f^{[2]} : b\ f^{[2]}(b_1, b_2) \in \Sigma \quad \Gamma \vdash e_1 : b_1 \quad \Gamma \vdash e_2 : b_2}{\Gamma \vdash e_1\ f^{[2]}\ e_2 : b}\texttt{Op2}$$

$$\frac{\Gamma(x) = b \quad (t_i)^{i \in \{\ldots n\}} \quad \{\Gamma \vdash e_i : t_i\}^{i \in \{\ldots n\}}}{\Gamma \vdash x\ \ (e_i)^{i \in \{\ldots n\}} : b}\texttt{FCall} \qquad \frac{\Gamma(x) = \mathsf{int}}{\Gamma \vdash x\texttt{++} : \mathsf{int}}\texttt{Inc} \qquad \frac{\Gamma(x) = \mathsf{int}}{\Gamma \vdash x\texttt{--} : \mathsf{int}}\texttt{Dec}$$

$$\frac{\Gamma(x) = b\ [\,] \quad \Gamma \vdash e : \mathsf{int}}{\Gamma \vdash x[e] : b}\texttt{Arr1} \qquad \frac{\Gamma(x) = *\ b[\,] \quad \Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash x[e_1][e_2] : b}\texttt{Arr2} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash e_1\ ?\ e_2 : e_3 : t}\texttt{If}$$

To indicate which terms must or can include a return statement, we define a more general judgment $\Gamma \vdash s : r$, where $r$ is defined as follows:

$$r \quad ::= \quad t \mid \bot$$

The $r$ types indicate return contexts: if a statement is well-formed with respect to the return context $\bot$, no return statements are permitted in it. If the return context is $t$, the judgment ensures that the rightmost leaf of the statement is a return statement of type $t$. For example, the definition body of a function with return type $t$, must be typed with $t$ as its return context.

Statements ($\Gamma \vdash s : t$)

$$\frac{\Gamma \vdash e : t_1}{\Gamma \vdash e : \bot}\texttt{EStat} \qquad \frac{\Gamma(x) = b \quad \Gamma \vdash e : b}{\Gamma \vdash x = e : \bot}\texttt{Assign} \qquad \frac{\Gamma(x) = b[\,] \quad \Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : b}{\Gamma \vdash x[e_1] = e_2 : \bot}\texttt{SetArr1}$$

$$\frac{\Gamma(x) = *b[\,] \quad \Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int} \quad \Gamma \vdash e_3 : b}{\Gamma \vdash x[e_1][e_2] = e_3 : \bot}\texttt{SetArr2} \qquad \frac{\Gamma \vdash e : \mathsf{int} \quad \Gamma \vdash s_1 : r \quad \Gamma \vdash s_2 : r}{\Gamma \vdash \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 : r}\texttt{IfS}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash s : \bot}{\Gamma \vdash \mathsf{while}\ (e_1)\ s : \bot}\texttt{While} \qquad \frac{\Gamma(x) = \mathsf{int} \quad \Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int} \quad \Gamma \vdash s : \bot}{\Gamma \vdash \mathsf{for}\ (x = e_1; e_2; x\texttt{++})\ s : \bot}\texttt{For} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathsf{return}\ (e) : t}\texttt{Ret}$$

$$\frac{\{c_i : \mathsf{int} \in \Sigma\}^{i\in\{\dots n\}} \quad \Gamma \vdash e : \mathsf{int} \qquad \{\Gamma \vdash s_i : \bot\}^{i\in\{\dots n\}} \quad \Gamma \vdash s : \bot}{\Gamma \vdash \left\{ \begin{array}{l} \mathsf{switch}\ (e)\{ \\ \quad (\mathsf{case}\ c_i : s_i\ \mathsf{break};)^{i\in\{\dots n\}} \\ \quad \mathsf{default} : s\} \end{array} \right\} : \bot}\texttt{Sw} \qquad \frac{\left\{\Gamma \cup (d_j)^{j\in\{\dots m\}} \vdash s_i\right\}^{i\in\{\dots n-1\}} : \bot \quad \Gamma \cup (d_j)^{j\in\{\dots m\}} \vdash s_n : r}{\Gamma \vdash \{(d_j)^{j\in\{\dots m\}}; (s_n)^{i\in\{\dots n\}}\} : r}\texttt{Blk}$$

Function definition ($\Gamma \vdash f$)

$$\frac{\Gamma \cup (a_i)^{i\in\{\dots n\}} \vdash \{(d_j)^{j\in\{\dots m\}}; (s_k)^{k\in\{\dots z\}}\} : b}{\Gamma \vdash (b\ f\ \ (a_i)^{i\in\{\dots n\}}\ \ \{(d_j)^{j\in\{\dots m\}}; (s_k)^{k\in\{\dots z\}}\})}\texttt{TFun}$$

Program ($\Gamma \vdash \overline{g}$)

$$\frac{}{\Gamma \vdash .}\texttt{Empty} \qquad \frac{\Gamma \vdash b\ f\ \ (a_i)^{i\in\{\dots n\}}\ \ s \quad \Gamma, b\ f\ \ (a_i)^{i\in\{\dots n\}} \vdash g}{\Gamma \vdash (b\ f\ (a_i)^{i\in\{\dots n\}}\ \ s); g}\texttt{ExtTop}$$

**Lemma 1.1 (Weakening)**

1. $\forall e, \Gamma, \Gamma', t.$ if $\Gamma \vdash e : t$ and $\mathtt{dom}\ \Gamma' \cap FV(e) = \emptyset$, then $\Gamma \cup \Gamma' \vdash e : t$.
2. $\forall s, \Gamma, \Gamma', r.$ if $\Gamma \vdash s : r$ and $\mathtt{dom}\ \Gamma' \cap FV(s) = \emptyset$, then $\Gamma \cup \Gamma' \vdash\ \ s : r$.

**Proof** Proofs are by induction on the height of the typing derivations. ■

# §2 Type System for the OCaml Fragment

Expressions ($\Gamma \vdash \hat{e} : \hat{t}$)

$$\frac{\hat{c} : \hat{b} \in \Sigma}{\Gamma \vdash \hat{c} : \hat{b}}\texttt{Const} \qquad \frac{\Gamma(\hat{x}) = \hat{t}}{\Gamma \vdash \hat{x} : \hat{t}}\texttt{Var}$$

$$\frac{\Gamma \vdash \hat{x} : (\hat{p}_1, \ldots, \hat{p}_n) \to \hat{b} \quad \{\Gamma \vdash \hat{e}_i : \hat{p}_i\}^{i \in \{1 \ldots n\}}}{\Gamma \vdash \hat{x}\ (e_i)^{i \in \{1 \ldots n\}} : \hat{b}}\texttt{FCall} \qquad \frac{\hat{f}^{(1)} : \hat{b}_1 \to \hat{b} \in \Sigma \quad \Gamma \vdash \hat{e} : \hat{b}_1}{\Gamma \vdash \hat{f}^{(1)}\ \hat{e} : \hat{b}}\texttt{Fun[1]} \qquad \frac{\hat{f}^{(2)} : \hat{b}_1 \to \hat{b}_2 \to \hat{b} \in \Sigma \quad \Gamma \vdash \hat{e}_1 : \hat{b}_1 \quad \Gamma \vdash \hat{e}_2 : \hat{b}_2}{\Gamma \vdash \hat{f}^{(2)}\ \hat{e}_1\ \hat{e}_2 : \hat{b}}\texttt{Fun(2)}$$

$$\frac{\hat{f}^{[2]} : \hat{b}_1 \to \hat{b}_2 \to \hat{b} \in \Sigma \quad \Gamma \vdash \hat{e}_1 : \hat{b}_1 \quad \Gamma \vdash \hat{e}_2 : \hat{b}_2}{\Gamma \vdash \hat{e}_1\ \hat{f}^{[2]}\ \hat{e}_2 : \hat{b}}\texttt{Fun[2]} \qquad \frac{\Gamma \vdash \hat{e}_1 : \texttt{bool} \quad \Gamma \vdash \hat{e}_2 : \hat{t} \quad \Gamma \vdash \hat{e}_3 : \hat{t}}{\Gamma \vdash \texttt{if}\ \hat{e}_1\ \texttt{then}\ \hat{e}_2\ \texttt{else}\ \hat{e}_3 : \hat{t}}\texttt{If}$$

$$\frac{\Gamma(\hat{x}) = \hat{t}\ \texttt{ref}}{\Gamma \vdash !\hat{x} : \hat{t}}\texttt{Ref} \qquad \frac{\Gamma \vdash \hat{e} : \texttt{int} \quad \Gamma(\hat{x}) = \hat{b}\ \texttt{array}}{\Gamma \vdash \hat{x}.(\hat{e}) : \hat{b}}\texttt{Arr1} \qquad \frac{\Gamma \vdash \hat{e}_1 : \texttt{int} \quad \Gamma \vdash \hat{e}_2 : \texttt{int} \quad \Gamma(\hat{x}) = \hat{b}\ \texttt{array array}}{\Gamma \vdash \hat{x}.(\hat{e}_1).(\hat{e}_2) : \hat{b}}\texttt{Arr2}$$

Statements ($\Gamma \vdash \hat{d} : \hat{t}$)

$$\frac{\Gamma \vdash \hat{e} : \hat{t}}{\Gamma \vdash \hat{e} : \hat{t}}\texttt{DExp} \qquad \frac{\Gamma \vdash \hat{d}_1 : \texttt{unit} \quad \Gamma \vdash \hat{d}_2 : \hat{t}}{\Gamma \vdash \hat{d}_1; \hat{d}_2 : \hat{t}}\texttt{SeqD} \qquad \frac{\Gamma \vdash \hat{e} : \hat{b} \quad \Gamma, \hat{x} : \hat{b} \vdash \hat{d} : \hat{t}}{\Gamma \vdash \left\{ \begin{array}{l} \texttt{let}\ \hat{x} : \hat{b} = \hat{e} \\ \texttt{in}\ \hat{d} \end{array} \right\} : \hat{t}}\texttt{Let}$$

$$\frac{\Gamma \vdash \hat{c} : \hat{b} \quad \Gamma, \hat{x} : \hat{b}\ \texttt{ref} \vdash \hat{d} : \hat{t}}{\Gamma \vdash \left\{ \begin{array}{l} \texttt{let}\ \hat{x} : \hat{b}\ \texttt{ref} = \texttt{ref}\ \hat{c} \\ \texttt{in}\ \hat{d} \end{array} \right\} : \hat{t}}\texttt{LetRef} \qquad \frac{\Gamma \vdash \hat{c_2} : \hat{b} \quad \Gamma, \hat{x} : \hat{b}\ \texttt{array} \vdash \hat{d} : \hat{t} \quad \Gamma \vdash \hat{c_1} : \texttt{int}}{\Gamma \vdash \left\{ \begin{array}{l} \texttt{let}\ \hat{x} : \hat{b}\ \texttt{array} = \\ \texttt{Array.make}\ \hat{c}_1\ \hat{c}_2\ \texttt{in}\ \hat{d} \end{array} \right\} : \hat{t}}\texttt{LetArr1}$$

$$\frac{\Gamma \vdash \hat{c_1} : \texttt{int} \quad \Gamma \vdash \hat{c_2} : \texttt{int} \quad \Gamma \vdash \hat{c_3} : \hat{b} \quad \Gamma, \hat{x} : \hat{b}\ \texttt{array array} \vdash \hat{d} : \hat{t}}{\Gamma \vdash \left\{ \begin{array}{l} \texttt{let}\ \hat{x} : \hat{b}\ \texttt{array array} = \\ \texttt{Array.make\_matrix}\ \hat{c}_1\ \hat{c}_2\ \hat{c}_3\ in\ \hat{d} \end{array} \right\} : \hat{t}}\texttt{LetArr2} \qquad \frac{\Gamma \vdash \hat{e} : \hat{b} \quad \Gamma(\hat{x}) = \hat{b}\ \texttt{ref}}{\Gamma \vdash (\hat{x} := \hat{e}) : \texttt{unit}}\texttt{Ref}$$

$$\frac{\Gamma(\hat{x}) = \hat{b}\ \texttt{array} \quad \Gamma \vdash \hat{e}_1 : \texttt{int} \quad \Gamma \vdash \hat{e}_2 : \hat{b}}{\Gamma \vdash \hat{x}.(\hat{e}_1) \leftarrow (\hat{e}_2) : \texttt{unit}}\texttt{SetArr1} \qquad \frac{\Gamma(\hat{x}) = \hat{b}\ \texttt{array array} \quad \Gamma \vdash \hat{e}_1 : \texttt{int} \quad \Gamma \vdash \hat{e}_2 : \texttt{int} \quad \Gamma \vdash \hat{e}_3 : \hat{b}}{\Gamma \vdash \hat{x}.(\hat{e}_1).(\hat{e}_2) \leftarrow (\hat{e}_3) : \texttt{unit}}\texttt{SetArr2}$$

$$\frac{\Gamma \vdash \hat{e} : \texttt{bool} \quad \Gamma \vdash \hat{d}_1 : \hat{t} \quad \Gamma \vdash \hat{d}_2 : \hat{t}}{\Gamma \vdash \texttt{if}\ \hat{e}\ \texttt{then}\ \hat{d}_1\ \texttt{else}\ \hat{d}_2 : \hat{t}}\texttt{IfD} \qquad \frac{\Gamma \vdash \hat{e} : \texttt{bool} \quad \Gamma \vdash \hat{d} : \texttt{unit}}{\Gamma \vdash \left\{ \begin{array}{l} \texttt{while}\ \hat{e} \\ \texttt{do}\ \hat{d}\ \texttt{done} \end{array} \right\} : \texttt{unit}}\texttt{While}$$

$$\frac{z \in \{\texttt{to}, \texttt{downto}\} \quad \Gamma \vdash \hat{e}_1 : \texttt{int} \quad \Gamma \vdash \hat{e}_2 : \texttt{int} \quad \Gamma, \hat{x} : \texttt{int} \vdash \hat{d} : \texttt{unit}}{\Gamma \vdash \left\{ \begin{array}{l} \texttt{for}\ \hat{x} : \texttt{int}\ = \hat{e}_1\ z\ \hat{e}_2 \\ \texttt{do}\ \hat{d}\ \texttt{done} \end{array} \right\} : \texttt{unit}}\texttt{For} \qquad \frac{\Gamma \vdash \hat{e} : \hat{b} \quad \hat{b} \neq \texttt{float} \quad \left\{\Gamma \vdash \hat{c_i} : \hat{b}\right\}^{i \in \{1 \ldots n\}} \quad \left\{\Gamma \vdash \hat{d_i} : \hat{t}\right\}^{i \in \{1 \ldots n\}} \quad \Gamma \vdash \hat{d} : \texttt{unit}}{\Gamma \vdash \left\{ \begin{array}{l} \texttt{match}\ \hat{e}\ \texttt{with} \\ \left(\left(\hat{c_i} \to \hat{d_i}\right)^{i \in \{\ldots n\}}\right. \\ \left.\ \mid \_ \to \hat{d}\right) \end{array} \right\} : \texttt{unit}}\texttt{Match}$$

Programs ($\Gamma \vdash \hat{s} : \hat{t}$)

$$\frac{\Gamma, (\hat{x}_i : \hat{p}_i)^{i \in \{\ldots n\}} \vdash \hat{d} : \hat{b}}{\Gamma \vdash \lambda\, (\hat{x}_i : \hat{p}_i)^{i \in \{\ldots n\}}\, .\, (\hat{d} : \hat{b}) : (p_i)^{i \in \{\ldots n\}} \to \hat{b}}\texttt{Toplevel} \qquad \frac{\begin{array}{c}\Gamma \vdash \hat{c} : \hat{b} \\ \Gamma, \hat{x} : \hat{b} \vdash \hat{s} : \hat{t}\end{array}}{\Gamma \vdash \left\{ \begin{array}{l}\texttt{let}\ \hat{x} : \hat{b} = \hat{c} \\ \texttt{in}\ \hat{s}\end{array} \right\} : \hat{t}}\texttt{LetS}$$

$$\frac{\begin{array}{c}\Gamma \vdash \hat{c} : \hat{b} \\ \Gamma, \hat{x} : \hat{b}\ \texttt{ref} \vdash \hat{s} : \hat{t}\end{array}}{\Gamma \vdash \left\{ \begin{array}{l}\texttt{let}\ \hat{x} : \hat{b}\ \texttt{ref} = \texttt{ref}\ \hat{c} \\ \texttt{in}\ \hat{s}\end{array} \right\} : \hat{t}}\texttt{LetRefS} \qquad \frac{\begin{array}{c}\Gamma \cup (\hat{x_i} : \hat{p_i})^{i \in \{m \ldots n\}} \vdash \hat{d} : \hat{b} \\ \Gamma \cup (\hat{x_i} : \hat{p_i})^{i \in \{m \ldots n\}}, \hat{x} : (\overline{p_n} \to \hat{b}) \vdash \hat{s} : \hat{t}\end{array}}{\Gamma \vdash \left\{ \begin{array}{l}\texttt{let}\ \hat{x}\ (x_i : \hat{p}_i)^{i \in \{\ldots n\}} : \hat{b} = \hat{d} \\ \texttt{in}\ \hat{s}\end{array} \right\} : \hat{t}}\texttt{LetFun}$$

$$\frac{\begin{array}{ll}\Gamma \vdash \hat{c_1} : \texttt{int} & \Gamma \vdash \hat{c_3} : \hat{b} \\ \Gamma \vdash \hat{c_2} : \texttt{int} & \Gamma, \hat{x} : \hat{b}\ \texttt{array array} \vdash \hat{s} : \hat{t}\end{array}}{\Gamma \vdash \left\{ \begin{array}{l}\texttt{let}\ \hat{x} : \hat{b}\ \texttt{array array} = \\ \texttt{Array.make\_matrix}\ \hat{c}_1\ \hat{c}_2\ \hat{c}_3\ \texttt{in}\ \hat{s}\end{array} \right\} : \hat{t}}\texttt{LetArr2S} \qquad \frac{\begin{array}{c}\Gamma \vdash \hat{c_1} : \texttt{int} \\ \Gamma \vdash \hat{c_2} : \hat{b} \\ \Gamma, \hat{x} : \hat{b}\ \texttt{array} \vdash \hat{s} : \hat{t}\end{array}}{\Gamma \vdash \left\{ \begin{array}{l}\texttt{let}\ \hat{x} : \hat{b}\ \texttt{array} = \\ \texttt{Array.make}\ \hat{c}_1\ \hat{c}_2 \\ \texttt{in}\ \hat{s}\end{array} \right\} : \hat{t}}\texttt{LetArr1S}$$

# §3 Properties of Offshoring Translation

**Lemma 3.1 (Basic properties of translation.)**

1. Translation relations $\{\!|\cdot|\!\} = (\cdot, \cdot)$, and $\langle\!|\cdot|\!\rangle = (\cdot, \cdot)$ are functions.
2. If $\{\!|\hat{d}|\!\} = (l, s)$, then $\texttt{dom}\ l \subseteq BV(d)$. If $\langle\!|\hat{s}|\!\rangle = (g, l)$, then $\texttt{dom}\ l \subseteq BV(s)$, $\texttt{dom}\ g \subseteq BV(s)$ and $\texttt{dom}\ l \cap \texttt{dom}\ g = \emptyset$.

**Proof** Both proofs are by induction on the type derivations and translation relation derivations, respectively. ■

# §4 Type Preservation

First, we define an operation $|\cdot|$ which translates variable declarations and function definitions to C type assignments.

$$\begin{array}{lcl} |\cdot| & = & \emptyset \\ |\ t\ x = \_; l| & = & \{x : t\} \cup |l| \\ |\ t\ f\ \ (t_i)^{i \in \{\ldots n\}}\ s\,; g| & = & \{t\ f\ \ (t_i)^{i \in \{\ldots n\}}\} \cup |g| \end{array}$$

We also define an operation $[\![\hat{\Gamma}]\!]$ which translates the OCaml variable and function environment, $\hat{\Gamma}$, into the corresponding C environment.

$$\begin{array}{lcl} [\![\cdot]\!] & = & \emptyset \\ [\![\hat{\Gamma}; \hat{x} : \hat{b}]\!] & = & [\![\hat{\Gamma}]\!] \cup \{x : [\![\hat{b}]\!]\} \\ [\![\hat{\Gamma}; (\hat{x_i} : \hat{p_i})^{i \in \{m \ldots n\}}, \hat{x} : (\overline{p_n} \to \hat{b})]\!] & = & [\![\hat{\Gamma}]\!] \cup \{[\![\hat{b}]\!]\ x([\![p_n]\!])\} \end{array}$$

Throughout the proofs the following assumption about free and bound variables in OCaml and C programs will be considered to hold:

**Assumption 1**
For any finite set of terms, no two bound variables in these terms have the same name.

Assumption 1 allows us to treat the nested scoping of OCaml let statements, where a variable can be rebound (e.g., `let x = 1 in let x = 2 in x` to `let x = 1 in let y = 2 in y` ), as morally equivalent to the flat scoping of C declarations and statements into which they are translated.

**Theorem 4.1 (Type preservation).** If $\hat{\Gamma} \vdash \hat{s} : \hat{a_n} \to \hat{b}$ and $\langle\!|\hat{s}|\!\rangle = (g, l)$, then $[\![\hat{\Gamma}]\!] \cup |l| \cup |g| \vdash g$.

**Proof** Proof is by induction on the height of typing derivation of programs.

**Case** $\Gamma \vdash \lambda\,(\hat{x}_i : \hat{p}_i)^{i\in\{\ldots n\}}\,.\,(\hat{d} : \hat{b}) : (p_i)^{i\in\{\ldots n\}} \to \hat{b}$**.** By inversion of typing for programs, $\Gamma, (\hat{x}_i : \hat{p}_i)^{i\in\{\ldots n\}} \vdash \hat{d} : \hat{b}$. By translation for programs, $\langle\!|\lambda\,(\hat{x}_i : \hat{p}_i)^{i\in\{\ldots n\}}\,.\,(\hat{d} : \hat{b}) : (p_i)^{i\in\{\ldots n\}} \to \hat{b}|\!\rangle = ([\![\hat{b}]\!]\ \mathsf{procedure}\ ([\![\hat{p_i}]\!]\ x_i)^{i\in\{\ldots n\}}\ \{l; s\}), \cdot)$ where $\{\!|\hat{d}|\!\}_\top = (l, s)$ We must show that $[\![\hat{\Gamma}]\!] \vdash [\![\hat{b}]\!]\ \texttt{procedure}\ ([\![\hat{p_i}]\!]\ x_i)^{i\in\{\ldots n\}}\ \{l; s\}$. This is true if $[\![\hat{\Gamma}]\!] \cup ([\![\hat{p_i}]\!]\ x_i)^{i\in\{\ldots n\}} \vdash \{l; s\} : [\![\hat{b}]\!]$ following the C `TFun` typing rule. We know this immediately by Lemma 4.1.3 and by definition of $[\![\hat{\Gamma}]\!]$.

**Case** $\hat{\Gamma} \vdash \texttt{let}\ \hat{x} : \hat{b} = \hat{c}\ \texttt{in}\ \hat{s} : \hat{t}$**.** We are also given $\langle\!|\texttt{let}\ \hat{x} = \hat{c}\ \texttt{in}\ \hat{s}|\!\rangle = (g, l)$. By inversion of typing for programs, we know that $\hat{\Gamma} \vdash \hat{c} : \hat{b}$ and $\hat{\Gamma}, \hat{x} : \hat{b} \vdash \hat{s} : \hat{t}$. Let $\langle\!|\hat{s}|\!\rangle = (g', l')$. Then we know by the translation function $\langle\!|\texttt{let}\, x : \hat{b} = \hat{c}\ \texttt{in}\ \hat{s}|\!\rangle = (g', ([\![\hat{b}]\!]\ x = c; l'))$. We must show that $[\![\hat{\Gamma}]\!] \cup |l| \cup |g| \vdash g$. By the induction hypothesis, $[\![\hat{\Gamma}, b\ x]\!] \cup |l'| \cup |g'| \vdash g'$. But now by definition of $[\![\hat{\Gamma}]\!]$, $[\![\hat{\Gamma}]\!] \cup \{b\ x\} \cup |l'| \cup |g'| \vdash g'$. However, what is given and what we know by the translation function imply $|g'| = |g|$ and $\{b\ x\} \cup |l'| = |l|$. Therefore, $[\![\hat{\Gamma}]\!] \cup |l| \cup |g| \vdash g$.

**Case** $\texttt{let}\ \hat{x}\ \ (x_i : \hat{p}_i)^{i\in\{\ldots n\}} : \hat{b} = \hat{d}\ \texttt{in}\ \hat{s} : \hat{t}$**.** We are also given $\langle\!|\texttt{let}\ \hat{x}\ \ (x_i : \hat{p}_i)^{i\in\{\ldots n\}} = \hat{d}\ \texttt{in}\ \hat{s}|\!\rangle = (g, l)$. By inversion of OCaml function typing, we know $\Gamma \cup (\hat{x_i} : \hat{p}_i)^{i\in\{m\ldots n\}} \vdash \hat{d} : \hat{b}$ and $\Gamma \cup (\hat{x_i} : \hat{p}_i)^{i\in\{m\ldots n\}}, \hat{x} : (\overline{p_n} \to \hat{b}) \vdash \hat{s} : \hat{t}$ By translation we know that $\langle\!|\texttt{let}\ \hat{x}\,(\hat{x_i} : \hat{p_i})^{i\in\{\ldots n\}} : \hat{b} = \hat{d}\ \texttt{in}\ \hat{s}|\!\rangle = ([\![\hat{b}]\!]x\,([\![p]\!]_i\ x_i)^{i\in\{\ldots n\}}\,\{l_d; s_d; \}; g', l')$ where $\langle\!|\hat{s}|\!\rangle = (g', l')$ and $\{\!|\hat{d}|\!\}_\top = (l_d, s_d)$. We are trying to show $[\![\hat{\Gamma}]\!] \cup |l'| \cup [\![\hat{b}]\!]\ x([\![p_n]\!]) \cup |g'| \vdash ([\![\hat{b}]\!]x\,([\![p]\!]_i\ x_i)^{i\in\{\ldots n\}}\,\{l_d; s_d; \}; g'$. By induction we have $[\![\hat{\Gamma}]\!] \cup [\![\hat{b}]\!]\ x([\![p_n]\!]) \cup$

$|l'| \cup |g'| \vdash g'$. Also by Lemma 4.1.3 and weakening, the function definition is well-typed, i.e.: $[\![\hat{\Gamma}]\!] \cup |l_d| \cup |s_d| \vdash ([\![\hat{b}]\!]x\,([\![p]\!]_i\ x_i)^{i\in\{\ldots n\}}\,\{l_d; s_d;\} : [\![\hat{b}]\!]$. By C `TFun`, $[\![\hat{\Gamma}]\!] \vdash ([\![\hat{b}]\!]x\,([\![p]\!]_i\ x_i)^{i\in\{\ldots n\}}\,\{l_d; s_d;\}$. Now by the C program typing $[\![\hat{\Gamma}]\!] \cup [\![\hat{b}]\!]\ x([\![p_n]\!]) \cup |l'| \cup |g'| \vdash ([\![\hat{b}]\!]x\,([\![p]\!]_i\ x_i)^{i\in\{\ldots n\}}\,\{l_d; s_d;\}; g'$.

**Case** $\hat{\Gamma} \vdash \texttt{let}\,x : \hat{b}\,\texttt{ref} = \texttt{ref}\,\hat{c}\,\texttt{in}\,\hat{s} : \hat{t}$**.** Similar to the previous `let` case.

**Case** $\hat{\Gamma} \vdash \texttt{let}\,x : \hat{b}\,\texttt{array} = \texttt{Array.make}\,\hat{c_1}\,\hat{c_2}\,\texttt{in}\,\hat{s} : \hat{t}$**.** Similar to previous case.

**Case** $\hat{\Gamma} \vdash \texttt{let}\,x : \hat{b}\,\texttt{array}\,\texttt{array} = \texttt{Array.make\_matrix}\,\hat{c}_1\ \hat{c}_2\ \hat{c}_3\ \texttt{in}\,\hat{s} : \hat{t}$**.** Similar to the previous case. ■

**Lemma 4.1 (Type preservation for expressions and statements)**

1. $\forall\hat{\Gamma}, \hat{e}, \hat{t}.$ if $\hat{\Gamma} \vdash \hat{e} : \hat{\tau},$ then $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}]\!] : [\![\hat{t}]\!]$
2. $\forall\hat{\Gamma}, \hat{d}.$ if $\hat{\Gamma} \vdash \hat{d} : \hat{t},$ and $\{\!|\hat{d}|\!\}_\perp = (l, s),$ then $[\![\hat{\Gamma}]\!] \vdash \{l, s\} : \perp$
3. $\forall\hat{\Gamma}, \hat{d}.$ if $\hat{\Gamma} \vdash \hat{d} : \hat{t},$ and $\{\!|\hat{d}|\!\}_\top = (l, s),$ then $[\![\hat{\Gamma}]\!] \vdash \{l, s\} : [\![\hat{t}]\!]$

**Proof** [Lemma 4.1.1] Proof is by induction on the height of the typing derivation of expressions.

**Case** $\hat{\Gamma} \vdash \hat{c} : \hat{t}$. Base case. Constants are translated to constants, the rules are virtually identical.

**Case** $\hat{\Gamma} \vdash \hat{x} : \hat{t}$. Base case. By inversion, of the typing judgment, we have $\hat{\Gamma}(\hat{x}) = \hat{t}$. Then, by definition of $[\![\cdot]\!]$, $[\![\hat{\Gamma}]\!](x) = [\![\hat{t}]\!]$, and therefore by the C `Var` rule $[\![\hat{\Gamma}]\!] \vdash x : [\![\hat{t}]\!]$.

**Case** $\hat{\Gamma} \vdash \hat{x}\ (\hat{e}_1, \ldots, \hat{e}_n) : \hat{b}$. By inversion we have $\hat{\Gamma} \vdash \hat{x} : (\hat{p_1}, \ldots, \hat{p_n}) \to \hat{b}$ thus $\hat{\Gamma}(\hat{x}) = (\hat{p_i})^{i\in\{\ldots n\}} \to \hat{b}$, and we have $\forall i.0 \le i \le n,\ \hat{\Gamma} \vdash \hat{e}_i : \hat{p}_i$. We apply the induction hypothesis to $\forall i.0 \le i \le n,\ \hat{\Gamma} \vdash \hat{e}_i : \hat{p}_i$ to obtain $\forall i.\, 0 \le i \le n$, $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}_i]\!] : [\![\hat{p}_i]\!]$. Then, by the C `Fcall` rule, we have $[\![\hat{\Gamma}]\!] \vdash x([\![\hat{e}_1]\!], \ldots, [\![\hat{e}_n]\!]) : [\![\hat{b}]\!]$.

**Case** $\hat{\Gamma} \vdash f^{(1)}\ \hat{e} : \hat{b}$**.** By inversion we have $f^{(1)} : \hat{b}_1 \to \hat{b} \in \Sigma$ and $\hat{\Gamma} \vdash \hat{e} : \hat{b}_1$. We apply the induction hypothesis to $\hat{\Gamma} \vdash \hat{e} : \hat{b}_1$ to obtain $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}]\!] : [\![\hat{b}_1]\!]$. Then, by the C `Op1` rule, we have $[\![\hat{\Gamma}]\!] \vdash f^{(1)}([\![\hat{e}]\!]) : [\![\hat{b}]\!]$.

**Case** $\hat{\Gamma} \vdash f^{(2)}\ \hat{e_1}\ \hat{e_2} : \hat{b}$. Similar to previous case.

**Case** $\hat{\Gamma} \vdash \hat{e_1}\ f^{[2]}\ \hat{e_2} : \hat{b}$. Similar to previous case.

**Case** $\hat{\Gamma} \vdash \texttt{if}\ \hat{e}_1\,\texttt{then}\,\hat{e_2}\,\texttt{else}\ \hat{e_3} : \hat{t}$**.** By inversion, followed by the induction hypothesis, we obtain $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}_1]\!] : \texttt{int}$, $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}_2]\!] : [\![\hat{t}]\!]$, $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}_3]\!] : [\![\hat{t}]\!]$. Then, by definition of $\vdash$, we have $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}_1]\!]\,\texttt{?}\,[\![\hat{e}_2]\!] : \hat{e}_3 : [\![\hat{t}]\!]$.

**Case** $\hat{\Gamma} \vdash \texttt{!}\hat{x} : \hat{t}$**.** Base case. By inversion we have $\hat{\Gamma}(\hat{x}) = \hat{t}\ \texttt{ref}$. By definition of $[\![\hat{\Gamma}]\!]$, $[\![\hat{\Gamma}]\!](x) = [\![\hat{t}]\!]$. Then, by the C `if` rule, we have $[\![\hat{\Gamma}]\!] \vdash x : [\![\hat{t}\ \texttt{ref}]\!]$.

**Case** $\hat{\Gamma} \vdash \hat{x}.(\hat{e}) : \hat{b}$**.** By inversion of $\vdash$, we obtain $\hat{\Gamma} \vdash \hat{e} : \texttt{int}$ and $\hat{\Gamma}(\hat{x}) = \hat{b}\ \texttt{array}$. We apply the induction hypothesis to $\hat{\Gamma} \vdash \hat{e} : \texttt{int}$ to obtain $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}]\!] :$

int. Also, by definition of $[\![\hat{\Gamma}]\!]$, we have $[\![\hat{\Gamma}]\!](x) = b[]$. Applying the `Arr1` rule of $\vdash$, we obtain $[\![\hat{\Gamma}]\!] \vdash [\![\hat{x}.(\hat{e})]\!] : [\![\hat{b}]\!]$.

**Case** $\hat{\Gamma} \vdash \hat{x}.(\hat{e}_1).(\hat{e}_2) : \hat{b}$**.** Similar to one-dimensional array indexing case. ■

**Proof** [Lemma 4.1.2] Proof is by induction on the height of the typing derivation of statements.

**Case** $\hat{\Gamma} \vdash \hat{e} : \hat{t}$**.** It is given that $\{\!|\hat{e}|\!\}_\perp = (l, s)$. Consider the case when by translation $\{\!|\hat{e}|\!\}_\perp = (\cdot, [\![\hat{e}]\!])$. We know $l = \cdot$ and $s = [\![\hat{e}]\!]$. By Lemma 4.1.1, $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}]\!] : [\![\hat{t}]\!]$, thus by the C `EStat` rule $[\![\hat{\Gamma}]\!] \vdash (\cdot, [\![\hat{e}]\!]) : \perp$.

**Case** $\hat{\Gamma} \vdash \hat{d_1}; \hat{d_2} : \hat{t}$**.** It is given that $\{\!|\hat{d_1}; \hat{d_2}|\!\}_\perp = (l, s)$. Having $\hat{\Gamma} \vdash \hat{d_1}; \hat{d_2} : \hat{t}$, by inversion we know that $\hat{\Gamma} \vdash \hat{d_1} : \texttt{unit}$ and $\hat{\Gamma} \vdash \hat{d_2} : \hat{t}$. Let $\{\!|\hat{d_1}|\!\}_\perp = (l_1, s_1)$ and $\{\!|\hat{d_2}|\!\}_\perp = (l_2, s_2)$. By translation, $\{\!|\hat{d_1}; \hat{d_2}|\!\}_\perp = (l_1; l_2, s_1; s_2)$. We know $l = l_1; l_2$ and $s = s_1; s_2$. We are trying to show $[\![\hat{\Gamma}]\!] \vdash \{l_1; l_2, s_1; s_2\} : \perp$. We apply the induction hypothesis to $\hat{\Gamma} \vdash \hat{d_1} : \texttt{unit}$ and $\hat{\Gamma} \vdash \hat{d_2} : \hat{t}$ to obtain $[\![\hat{\Gamma}]\!] \vdash \{l_1, s_1\} : \perp$ and $[\![\hat{\Gamma}]\!] \vdash \{l_2, s_2\} : \perp$. By inversion of the C `Blk` rule for $[\![\hat{\Gamma}]\!] \vdash \{l_1, s_1\} : \perp$ and $[\![\hat{\Gamma}]\!] \vdash \{l_2, s_2\} : \perp$, we have $[\![\hat{\Gamma}]\!] \cup |l_1| \vdash s_1 : \perp$ and $[\![\hat{\Gamma}]\!] \cup |l_2| \vdash s_2 : \perp$. By Lemma 3.1.2 we know $\texttt{dom}\ l_1 \subseteq BV(d_1)$ and $\texttt{dom}\ l_2 \subseteq BV(d_2)$. Now by Assumption 1, $\texttt{dom}\ l_1 \cap \texttt{dom}\ l_2 = \emptyset$. We can conclude, by weakening that $[\![\hat{\Gamma}]\!] \cup |l_1| \cup |l_2| \vdash s_1 : \perp$ and $[\![\hat{\Gamma}]\!] \cup |l_1| \cup |l_2| \vdash s_2 : \perp$. The final result, $[\![\hat{\Gamma}]\!] \vdash \{l_1; l_2, s_1; s_2\} : \perp$, follows by the C rule `Blk`.

**Case** $\hat{\Gamma} \vdash \hat{x} := \hat{e} : \texttt{unit}$**.** It is given that $\{\!|\hat{x} := \hat{e}|\!\}_\perp = (l, s)$. By inversion, we know that $\hat{\Gamma} \vdash \hat{e} : \hat{b}$ and $\hat{\Gamma}(\hat{x}) = b\ \texttt{ref}$. Let $\{\!|\hat{x} := \hat{e}|\!\}_\perp = (\cdot, x = [\![\hat{e}]\!])$. By Lemma 4.1.1, we know that $[\![\hat{\Gamma}]\!] \vdash e : [\![\hat{b}]\!]$. By definition of $[\![\hat{\Gamma}]\!]$, we know $[\![\hat{\Gamma}]\!](x) = [\![\hat{b}]\!]$. Therefore, by the C `Assign` rule, $[\![\hat{\Gamma}]\!] \vdash x = [\![\hat{e}]\!] : \perp$.

**Case** $\hat{\Gamma} \vdash \texttt{let}\, x : \hat{b} = \hat{e}\, \texttt{in}\, \hat{d} : \hat{t}$**.** It is given that $\{\!|\texttt{let}\, x = \hat{e}\, \texttt{in}\, \hat{d}|\!\}_\perp = (l, s)$. By inversion of OCaml typing, we know that $\hat{\Gamma} \vdash \hat{e} : \hat{b}$ and $\hat{\Gamma}, \hat{x} : \hat{b} \vdash \hat{d} : \hat{t}$. Let $\{\!|\hat{d}|\!\}_\perp = (l', s')$. Then by translation we know $\{\!|\texttt{let}\, x : \hat{b} = \hat{e}\, \texttt{in}\, \hat{d}|\!\}_\perp = (([\![\hat{b}]\!]\ x; l'), (x = [\![\hat{e}]\!]; s'))$. By Lemma 4.1.1, we know that $[\![\hat{\Gamma}]\!] \vdash e : [\![\hat{b}]\!]$. By the C `Assign` rule, we know $[\![\hat{\Gamma}]\!] \vdash x = e : \perp$. We must show that $[\![\hat{\Gamma}]\!] \vdash (([\![\hat{b}]\!]\ x; l'), (x = [\![\hat{e}]\!]; s')) : \perp$. By the induction hypothesis, $[\![\hat{\Gamma}, \hat{x} : \hat{b}]\!] \vdash (l', s') : \perp$. The final result follows by the C `Blk` rule.

**Case** $\hat{\Gamma} \vdash \texttt{let}\, x : \hat{b}\, \texttt{ref} = \texttt{ref}\, \hat{c}\, \texttt{in}\, \hat{d} : \hat{t}$**.** Similar to previous case.

**Case** $\hat{\Gamma} \vdash \texttt{let}\, x : \hat{b}\, \texttt{array} = \texttt{Array.make}\, \hat{c_1}\, \hat{c_2}\, \texttt{in}\, \hat{d} : \hat{t}$. Similar to previous case.

**Case** $\hat{\Gamma} \vdash \texttt{let}\, x : \hat{b}\, \texttt{array}\, \texttt{array} = \texttt{Array.make\_matrix}\, \hat{c_1}\ \hat{c_2}\ \hat{c_3}\ \texttt{in}\ \hat{d} : \hat{t}$**.** Similar to previous case.

**Case** $\hat{\Gamma} \vdash \hat{x}.(\hat{e_1}) \leftarrow \hat{e_2} : \texttt{unit}$. It is given that $\{\!|\hat{x}.(\hat{e_1}) \leftarrow \hat{e_2}|\!\}_\perp = (l, s)$. By inversion we know $\hat{\Gamma}(x) = \hat{b}\ \texttt{array}$, $\hat{\Gamma} \vdash \hat{e_1} : \texttt{int}$ and $\hat{\Gamma} \vdash \hat{e_2} : \hat{b}$. Let $\{\!|\hat{x}.(\hat{e_1}) \leftarrow$

$\hat{e}_2\}_\perp = (., x[[\hat{e_1}]] = [\![\hat{e_2}]\!])$. We apply the definition of $\{|\hat{\Gamma}|\}$ to $\hat{\Gamma} \vdash \hat{e}_1 : \texttt{int}$ and Lemma 4.1.1 to $\hat{\Gamma} \vdash \hat{e}_1 : \texttt{int}$ and $\hat{\Gamma} \vdash \hat{e}_2 : \hat{b}$ to obtain $[\![\hat{\Gamma}]\!](x) = b[]$, $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}_1]\!] : \texttt{int}$ and $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}_2]\!] : b$ respectively. By the C rule for array assignment `SetArr1`, we now have: $[\![\Gamma]\!] \vdash x[[\![\hat{e_1}]\!]] = [\![(\hat{e}_2)]\!] : \perp$.

**Case** $\hat{\Gamma} \vdash \hat{x}.(\hat{e}_1).(\hat{e}_2) \leftarrow \hat{e}_3 : \texttt{unit}$**.** Similar to previous case.

**Case** $\hat{\Gamma} \vdash \texttt{if}\,\hat{e}_1\,\texttt{then}\,\hat{d}_1\,\texttt{else}\,\hat{d}_2 : \hat{t}$**.** By inversion, we know that $\hat{\Gamma} \vdash \hat{e} : \texttt{bool}$, $\hat{\Gamma} \vdash \hat{d_1} : \hat{t}$ and $\hat{\Gamma} \vdash \hat{d_2} : \hat{t}$. By translation, $\{|\texttt{if}\,\hat{e}_1\,\texttt{then}\,\hat{d}_1\,\texttt{else}\,\hat{d}_2|\}_\perp = (., \mathsf{if}\ ([\![\hat{e}]\!])\{l_1; s_1\}\ \mathsf{else}\ \{l_2; s_2\})$ where $\{|\hat{d_1}|\}_\perp = (l_1; s_1)$ and $\{|\hat{d_2}|\}_\perp = (l_2; s_2)$ From Lemma 4.1.1, $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}]\!] : \texttt{int}$. By induction, $[\![\hat{\Gamma}]\!] \vdash (l_1, s_1) : \perp$ and $[\![\hat{\Gamma}]\!] \vdash (l_2, s_2) : \perp$. Then by the C `Ifs` rule, $[\![\hat{\Gamma}]\!] \vdash (., \mathsf{if}\ ([\![\hat{e}]\!])\{l_1; s_1\}\ \mathsf{else}\ \{l_2; s_2\}) : \perp$ .

**Case** $\hat{\Gamma} \vdash \texttt{while}\,\hat{e}\,\texttt{do}\,\hat{d}\,\texttt{done} : \texttt{unit}$**.** By inversion we know that $\hat{\Gamma} \vdash \hat{e} : \texttt{bool}$ and $\hat{\Gamma} \vdash \hat{d} : \hat{t}$. By translation, $\{|\texttt{while}\,\hat{e}\,\texttt{do}\,\hat{d}\,\texttt{done}|\}_\perp = (., \mathsf{while}\ ([\![\hat{e}]\!])\{l'; s'\})$ where $\{|\hat{d}|\}_\perp = (l; s')$ . From Lemma 4.1.1, $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}]\!] : \texttt{int}$. By induction, $[\![\hat{\Gamma}]\!] \vdash (l'; s') : \perp$ Then by the C `While` rule, $[\![\hat{\Gamma}]\!] \vdash (., \mathsf{while}\ ([\![\hat{e}]\!])\{l'; s'\}) : \perp$.

**Case** $\hat{\Gamma} \vdash \texttt{for}\,\hat{x} = \hat{e}_1\,\{\texttt{to,downto}\}\,\hat{e}_2\,\texttt{do}\,\hat{d}\,\texttt{done} : \texttt{unit}$**.** By inversion, we know that $\hat{\Gamma} \vdash \hat{e_1} : \texttt{int}$, $\hat{\Gamma} \vdash \hat{e_2} : \texttt{int}$ and $\hat{\Gamma}, \hat{x} : \texttt{int} \vdash \hat{d} : \hat{t}$. By translation, $\{|\texttt{for}\,\hat{x} = \hat{e}_1\,\texttt{to}\,\hat{e}_2\,\texttt{do}\,\hat{d}\,\texttt{done}|\}_\perp = (\mathsf{int}\ x, \mathsf{for}\ (x = [\![\hat{e_1}]\!]; x\texttt{<=}[\![\hat{e_2}]\!]; x\texttt{++})\{l'; s'\})$ where $\{|\hat{d}|\} = (l'; s')$. From Lemma 4.1.1, $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e_1}]\!] : \texttt{int}$ and $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e_2}]\!] : \texttt{int}$. Also, by definition of $[\![\hat{\Gamma}]\!]$, we have $[\![\hat{\Gamma}]\!](x) = \texttt{int}$. By induction, $[\![\hat{\Gamma}]\!] \vdash (l, s) : \perp$. Then by the C `For` rule, $[\![\hat{\Gamma}]\!] \vdash (\mathsf{int}\ x, \mathsf{for}\ (x = [\![\hat{e_1}]\!]; x\texttt{<=}[\![\hat{e_2}]\!]; x\texttt{++})\{l'; s'\}) : \perp$.

**Case** $\texttt{match}\ \hat{e}\ \texttt{with}(\left(\hat{c}_i \rightarrow \hat{d}_i\right)^{i\in\{...n\}} \mid \_ \leftarrow \hat{d}) : \texttt{unit}$**.** By inversion, we know that $\hat{\Gamma} \vdash \hat{e} : \hat{b}$ where $\hat{b}$ is not a `float`. Also, $\hat{\Gamma} \vdash \overline{\hat{c_n} : \hat{b}}$ and $\hat{\Gamma} \vdash \overline{\hat{d_n} : \hat{t}}$ and $\hat{\Gamma} \vdash \hat{d} : \texttt{unit}$. By translation, $\{|\texttt{match}\ \hat{e}\ \texttt{with}(\left(\hat{c}_i \rightarrow \hat{d}_i\right)^{i\in\{...n\}} \mid \_ \leftarrow \hat{d})|\}_\perp =$ $\mathsf{switch}\ ([\![e]\!])\{\overline{\mathsf{case}\ c_n : \{l'_n; s'_n; \mathsf{break}\}};\ \mathsf{default} : \{l'; s'\}\}$ where $\{|\hat{d_n}|\}_\perp = \overline{(l'_n, s'_n)}$ and $\{|\hat{d}|\}_\perp = (l', s')$. From Lemma 4.1.1, $[\![\hat{\Gamma}]\!] \vdash e : [\![\hat{b}]\!]$ and $[\![\Gamma]\!] \vdash \overline{c_n : \texttt{int}}$. By induction, $[\![\hat{\Gamma}]\!] \vdash \overline{(l'_n, s'_n) : \perp}$ and $[\![\hat{\Gamma}]\!] \vdash (l', s') : \perp$. Then by the C `Switch` rule $[\![\hat{\Gamma}]\!] \vdash \mathsf{switch}\ ([\![e]\!])\{\overline{\mathsf{case}\ c_n : \{l'_n; s'_n; \mathsf{break}\}};\ \mathsf{default} : \{l'; s'\}\} : \perp$ . ■

**Proof** [Lemma 4.1.3] Proof is by induction on the height of the typing derivation of statements. The proof is similar to the one for Lemma 4.1.2. Cases where $\{|\cdot|\}$ is undefined follow trivially. We concentrate on the remaining cases.

**Case** $\hat{\Gamma} \vdash \hat{e} : \hat{t}$**.** It is given that $\{|\hat{e}|\}_\top = (l, s)$. Consider the case when by translation $\{|\hat{e}|\}_\top = (\cdot, \mathsf{return}\ [\![\hat{e}]\!])$. We know $l = \cdot$ and $s = \mathsf{return}\ [\![\hat{e}]\!]$. By Lemma 4.1.1, $[\![\hat{\Gamma}]\!] \vdash [\![\hat{e}]\!] : [\![\hat{t}]\!]$, thus by the C `Ret` rule $[\![\hat{\Gamma}]\!] \vdash \{\mathsf{return}\ [\![\hat{e}]\!];\} : [\![\hat{t}]\!]$.

**Case** $\hat{\Gamma} \vdash \hat{d_1}; \hat{d_2} : \hat{t}$**.** It is given that $\{\!|\hat{d_1}; \hat{d_2}|\!\}_\top = (l, s)$. Having $\hat{\Gamma} \vdash \hat{d_1}; \hat{d_2} : \hat{t}$, by inversion we know that $\hat{\Gamma} \vdash \hat{d_1} : \texttt{unit}$ and $\hat{\Gamma} \vdash \hat{d_2} : \hat{t}$. Let $\{\!|\hat{d_1}|\!\}_\bot = (l_1, s_1)$ and $\{\!|\hat{d_2}|\!\}_\top = (l_2, s_2)$. By translation, $\{\!|\hat{d_1}; \hat{d_2}|\!\}_\top = (l_1; l_2, s_1; s_2)$. We know $l = l_1; l_2$ and $s = s_1; s_2$. We are trying to show $[\![\hat{\Gamma}]\!] \vdash \{l_1; l_2, s_1; s_2\} : [\![\hat{t}]\!]$. We apply Lemma 4.1.2 to $\hat{\Gamma} \vdash \hat{d_1} : \texttt{unit}$ and the induction hypothesis to $\hat{\Gamma} \vdash \hat{d_2} : \hat{t}$ to obtain $[\![\hat{\Gamma}]\!] \vdash \{l_1, s_1\} : \bot$ and $[\![\hat{\Gamma}]\!] \vdash \{l_2, s_2\} : [\![\hat{t}]\!]$. By inversion of the C `Blk` rule for $[\![\hat{\Gamma}]\!] \vdash \{l_1, s_1\} : \bot$ a nd $[\![\hat{\Gamma}]\!] \vdash \{l_2, s_2\} : [\![\hat{t}]\!]$, we have $[\![\hat{\Gamma}]\!] \cup |l_1| \vdash s_1 : \bot$ and $[\![\hat{\Gamma}]\!] \cup |l_2| \vdash s_2 : [\![t]\!]$. By Lemma 3.1.2 we know $\texttt{dom}\ l_1 \subseteq BV(d_1)$ and $\texttt{dom}\ l_2 \subseteq BV(d_2)$. Now by Assumption 1, $\texttt{dom}\ l_1 \cap \texttt{dom}\ l_2 = \emptyset$. We can conclude, by weakening that $[\![\hat{\Gamma}]\!] \cup |l_1| \cup |l_2| \vdash s_1 : \bot$ and $[\![\hat{\Gamma}]\!] \cup |l_1| \cup |l_2| \vdash s_2 : [\![t]\!]$. The final result, $[\![\hat{\Gamma}]\!] \vdash \{l_1; l_2, s_1; s_2\} : [\![\hat{t}]\!]$, follows by the C rule `Blk`. **Case** $\hat{\Gamma} \vdash \texttt{let}\, x : \hat{b} = \hat{e}\, \texttt{in}\, \hat{d} : \hat{t}$**.** It is given that $\{\!|\texttt{let}\, x = \hat{e}\, \texttt{in}\, \hat{d}|\!\}_\top = (l, s)$. By inversion of OCaml typing, we know that $\hat{\Gamma} \vdash \hat{e} : \hat{b}$ and $\hat{\Gamma}, \hat{x} : \hat{b} \vdash \hat{d} : \hat{t}$. Let $\{\!|\hat{d}|\!\}_\top = (l', s')$. Then by translation we know $\{\!|\texttt{let}\, x : \hat{b} = \hat{e}\, \texttt{in}\, \hat{d}|\!\}_\top = (([\![\hat{b}]\!]\ x; l'), (x = [\![\hat{e}]\!]; s'))$. By Lemma 4.1.1, we know that $[\![\hat{\Gamma}]\!] \vdash e : [\![\hat{b}]\!]$. By the C `Assign` rule, we know $[\![\hat{\Gamma}]\!] \vdash x = e : \bot$. We must show that $[\![\hat{\Gamma}]\!] \vdash (([\![\hat{b}]\!]\ x; l'), (x = [\![\hat{e}]\!]; s')) : [\![\hat{t}]\!]$. By the induction hypothesis, $[\![\hat{\Gamma}, \hat{x} : \hat{b}]\!] \vdash (l', s') : [\![\hat{t}]\!]$. The final result follows by the C `Blk` rule.

**Case** $\hat{\Gamma} \vdash \texttt{let}\, x : \hat{b}\, \texttt{ref} = \texttt{ref}\, \hat{c}\, \texttt{in}\, \hat{d} : \hat{t}$**.** Similar to previous case.

**Case** $\hat{\Gamma} \vdash \texttt{let}\, x : \hat{b}\, \texttt{array} = \texttt{Array.make}\, \hat{c_1}\, \hat{c_2}\, \texttt{in}\, \hat{d} : \hat{t}$. Similar to previous case.

**Case** $\hat{\Gamma} \vdash \texttt{let}\, x : \hat{b}\, \texttt{array}\, \texttt{array} = \texttt{Array.make\_matrix}\, \hat{c_1}\ \hat{c_2}\ \hat{c_3}\ \texttt{in}\ \hat{d} : \hat{t}$**.** Similar to previous case. ∎

# §5 Benchmark Data

**Timing.** The execution times from which our data is derived are collected using MetaOCaml's standard library function `Trxtime.timenew`. This function repeatedly executes a program until the cumulative execution time exceeds 1 second and reports the number of iterations and the average execution time per iteration.

**Offshoring vs. Byte Code Compiler.** Table 1 displays measurements for offshoring with both `gcc` and Intel's `icc`. The columns are computed as follows: **Unstaged** is the execution time of the unstaged version of a program. **Generate** reports code generation times. Generation is considered the first stage in a two-stage computation. The second stage is called Staged Run and will

**Table 1** Speedups and Break-Even Points from Offshoring to C

| Name | Unstaged (ms) | Generate (ms) | Compile (ms) | Staged Run ($\mu$s) | Speedup | **Speedup'** | **BEP** |
|---|---|---|---|---|---|---|---|
| forward | 0.20 | 1.1 | 34. | 0.37 | 530x | **20.x** | **180** |
| gib | 0.13 | 0.26 | 19. | 0.12 | 990x | **5.3x** | **170** |
| ks | 5.2 | 36. | 8300. | 1.4 | 3700x | **69.x** | **1600** |
| lcs | 5.5 | 46. | 6400. | 5.1 | 1100x | **24.x** | **1100** |
| obst | 4.2 | 26. | 5300. | 4.6 | 910x | **22.x** | **1300** |
| opt | 3.6 | 66. | 8700. | 3.4 | 1100x | **44.x** | **2500** |

Using GNU's `gcc`

| Name | Unstaged (ms) | Generate (ms) | Compile (ms) | Staged Run ($\mu$s) | Speedup | **Speedup'** | **BEP** |
|---|---|---|---|---|---|---|---|
| forward | 0.20 | 1.1 | 33.0 | 0.35 | 580x | **21.x** | **170** |
| gib | 0.13 | 0.26 | 20.0 | 0.098 | 1200x | **6.4x** | **180** |
| ks | 5.2 | 36. | 8100. | 1.5 | 3500x | **66.x** | **1600** |
| lcs | 5.5 | 46. | 6500. | 5.3 | 1000x | **25.x** | **1200** |
| obst | 4.2 | 26. | 5400. | 4.5 | 940x | **23.x** | **1300** |
| opt | 3.6 | 66. | 9300. | 3.3 | 1100x | **46.x** | **2600** |

Using Intel's `icc`

be described shortly. **Compile** is the time needed to translate the generated program and compile it with `gcc` (or `icc` in the second table), and dynamically load the resulting binary. **Staged Run** reports the C program execution time, including marshalling. The binary is generated by calling the C compiler with the flags `-g -O2`. **Speedup** is computed as Unstaged divided by Staged Run. **Speedup'** is the ratio between Speedup with staging (without offshoring) and staging with offshoring. **BEP** is the break-even point.

**Offshoring vs. OCaml Native Compiler.** Table 2 displays the data for Figure 5. The columns are computed as follows: **OCamlOpt** is the time for executing programs compiled with the `ocamlopt` compiler with the options `-unsafe -inline 50`. **Tweaked** are execution times for programs hand-optimized post-generation. The same compiler options as in **OCamlOpt** were used. **Best GCC** (**Best ICC**) is the execution time for generated and offshored code with the best performing optimization level (`-O[0-4]`). **Speedup** is the ratio of the **OCamlOpt** times to the **Best GCC** (**Best ICC**) times, and **Speedup"** is the ratio of the **Tweaked** execution times **Best GCC** (**Best ICC**).

**Table 2** Speed of offshored vs. OCaml native compiled code

| Name | OCamlOpt ($\mu$s) | Tweaked ($\mu$s) | Best GCC ($\mu$s) | **Speedup** | **Speedup"** |
|---|---|---|---|---|---|
| forward | 0.29 | 0.25 | 0.13 | **2.2x** | **2.1x** |
| gib | 0.017 | 0.0095 | 0.0096 | **1.7x** | **0.95x** |
| ks | 23. | 0.61 | 1.0 | **23.x** | **0.59x** |
| lcs | 24. | 3.1 | 1.2 | **20.x** | **3.0x** |
| obst | 33. | 10. | 2.7 | **12.x** | **3.9x** |
| opt | 24. | 4.9 | 2.8 | **8.3x** | **1.7x** |

Using GNU's `gcc`

| Name | OCamlOpt ($\mu$s) | Tweaked ($\mu$s) | Best ICC ($\mu$s) | **Speedup** | **Speedup"** |
|---|---|---|---|---|---|
| forward | 0.29 | 0.25 | 0.13 | **2.2x** | **1.8x** |
| gib | 0.017 | 0.0095 | 0.0091 | **1.8x** | **1.1x** |
| ks | 23. | 0.61 | 0.061 | **380.x** | **10.x** |
| lcs | 24. | 3.1 | 1.1 | **22.x** | **2.2x** |
| obst | 33. | 10. | 2.9 | **11.x** | **3.6x** |
| opt | 24. | 4.9 | 3.1 | **7.8x** | **1.6x** |

Using Intel's `icc`