

Implicitly Heterogeneous Multi-Stage Programming^{*}

(Extended Version)

Jason Eckhardt, Roumen Kaiabachev, Emir Pašalić, Kedar Swadi
and Walid Taha

Rice University, Houston, TX 77005, USA

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.

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.

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. 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 (c.f. [19, 1]).

^{*} Supported by 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.”

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 in which the programmer may wish to take advantage of the capability 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 heterogeneous quotation mechanisms also introduce 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 unfamiliar type systems.

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. The typing infrastructure needed is likely to be far 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. Those details are supplied by the meta-language designer once and for all and invoked by the programmer through the familiar interface used to execute generated code. This is achieved by the language implementers providing 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. The design 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). This requires being explicit about the details of the type system for the target language as well as the source language, but is manageable because both subsets are syntactically limited. 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 world to the C world 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.

Due to space limitations, the paper only formalizes the key concepts that illustrate the main ideas. Auxiliary definitions and proofs can be found in an extended version of the paper available online [4].

2 Multi-Stage Programming

MSP languages [20, 16] provide three high-level constructs that allow the programmer to break down computations into distinct stages. These constructs can be used for the construction, combination, and execution of 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 (cf. [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 un-staged 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. The mapping from base to target is only described informally. 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 code they produce amenable to translation. (Please see Appendix C for full definition.) The 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`, `char` and `double`.
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 MetaOCaml 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:

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

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:

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

Fig. 1. Grammar for the C target

1. All declarations are initialized. This restriction is caused by 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.
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 A).

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,

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

Fig. 2. OCaml Subset Grammar

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.

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 `bool`, `int`, `char`, and `float` map to C `int`, `char` and `double`, (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 `array` and `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:

<i>Base</i>	$\hat{b} \in \{\text{int}, \text{bool}, \text{float}, \text{char}\}$
<i>Reference</i>	$\hat{r} ::= \hat{b} \text{ ref}$
<i>Array</i>	$\hat{a} ::= \hat{b} \text{ array} \mid \hat{b} \text{ array array}$
<i>Argument</i>	$\hat{p} ::= \hat{b} \mid \hat{a}$
<i>Variables</i>	$\hat{t} ::= \hat{b} \mid \hat{r} \mid \hat{a}$
<i>Function</i>	$\hat{u} ::= (\hat{p}_0, \dots, \hat{p}_n) \rightarrow \hat{b}$

3.4 Syntax for Base Language

The syntax of the source subset is presented in Figure 2. Semantically, this subset represents the first-order subset of OCaml with arrays and reference cells. We point out the most interesting syntactic features of the OCaml subset:

1. 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.
2. Assignments in C are represented by updating OCaml references or arrays.
3. We use a special syntactic form to indicate a top level entry function (analogous to C `main`).

3.5 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 when offshoring should be used and whether 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`.

3.6 Marshalling and Dynamic Linking

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

In our implementation of the translation described above, we generate a marshalling 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, OCaml 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 has been produced, the marshalling code is added, and 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. Here we give a brief outline of the technical development.

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 B). Similarly, the type system for C programs is defined by a judgment $\Gamma \vdash g$ (Appendix A). We also provide the definition of translation functions (Appendix C). For example, $\langle \hat{s} \rangle = (g_1, \dots, g_n, l_1, \dots, l_m)$ translates a top-level OCaml program into a set of C variable declarations and function definitions, and $\llbracket \hat{\Gamma} \rrbracket$ translates the OCaml variable and function environment, $\hat{\Gamma}$, into the corresponding C environment.

Any valid term in the base language translates to a valid one in the target language. We define an operation $|\cdot|$ on variable declarations and function definitions that translates them into a C type environment (Appendix D).

Theorem 1 (Type Preservation). *If $\hat{\Gamma} \vdash \hat{s} : \hat{a}_n \rightarrow \hat{b}$ and $\langle \hat{s} \rangle = (g, l)$, then $\llbracket \hat{\Gamma} \rrbracket \cup |l| \cup |g| \vdash g$.*

Proof. By induction over the height of first derivation. The details of the proof are presented in the extended version of the paper [4]. \square

5 Effect on Performance Characteristics

This section summarizes some empirical measurements that we have gathered to evaluate the performance impact of offshoring. The questions we wanted 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 how dynamic programming algorithms can be staged [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 function. This is not an example of dynamic programming, but gives rise to the same technical problems while staging. Specialization is for $n = 25$.
- **ks**, the 0/1 knapsack problem. Specialization is for size 32 (number of items).

Name	Unstaged (ms)	Generate (ms)	Compile (ms)	Staged Run (μ 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 (μ 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

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

- **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.¹

5.2 Comparing Offshoring with the OCaml Byte Code

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.

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. All times are averages, and are reported as

¹ **Platform specs.** 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.

in milliseconds (ms) unless otherwise stated. We measure execution times 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. **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 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, i.e., the number of executions for a piece of code after which the initial overhead of offshoring (generation and compilation) pays off. For example, we need at least 180 uses of **forward** compiled using `gcc` before it is cheaper to use the staged version than to just use the unstaged one.

In general, the results obtained for offshoring to C are encouraging. The ratio between speedups (**Speedup’**) is always greater than 1. But it should be noted that the BEP’s are higher than for 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.

Table 2 displays the numbers for the `gcc` and `icc` compilers. Each program is run multiple times externally using the same number of iterations obtained from the measurements for the **Unstaged Run** in Figure 1. The Unix `time` command is used to measure the time for each program. This time is then divided by the number of iterations to obtain an average execution time per program. The columns are computed as follows: **Marshalling Overhead(G)** and **Marshalling Overhead(I)** show the difference in microseconds between

Name	Marshalling (G) Overhead (μs)	Percentage (G)	Marshalling (I) Overhead (μs)	Percentage (I)
forward	0.19	50. %	0.22	63. %
gib	0.11	91. %	0.089	91. %
ks	0.37	26. %	1.4	96. %
lcs	3.8	74. %	4.2	79. %
obst	0.61	13. %	1.5	34. %
opt	0.61	17. %	0.24	7.4 %

Table 2. Marshalling Overhead for `gcc` and Intel’s `icc` Compiled Offshored Code

Name	OCamlOpt (μ s)	Tweaked (μ s)	Best GCC (μ 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 (μ s)	Tweaked (μ s)	Best ICC (μ 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

Table 3. Speed of Offshored vs. OCaml Native Compiled Code

running each function from within MetaOCaml and running it externally using `gcc` and `icc` respectively. **Percentage(G)** and **Percentage(I)** show the marshalling overhead as a percentage of the total time spent on the marshalled computation.

In the `forward` example, we marshal a 7-element integer array. In `gib`, we marshal only two integers, and thus have a low marshalling overhead. But because the total computation time in this benchmark small, the marshalling overhead is high. In `ks`, we marshal a 32-element integer array, and in `lcs` benchmark, we marshal two 25- and 34-element character arrays, which accounts for the high marshalling overhead in these benchmarks. Since both these benchmarks perform a significant amount of computation, the proportion of time spent marshalling is, however, lower than `gib` or `forward`. Similarly, `obst` and `opt` have significant marshalling overheads since they must marshal 15-element floating-point and 18-element integer arrays.

While the percentages given by both `gcc` and `icc` are comparable, there is one benchmark `ks` which shows a large difference. This can be explained by the fact that `icc` was able to optimize much better than `gcc`, reducing computation time to the point that most of the time was spent in marshalling.

The data indicates that marshalling overhead is significant in this benchmark. 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 Comparing Offshoring with the 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?

Table 3 displays measurements for execution times of the generated programs when executed using the native-code compiler for OCaml and the best runtime when running the result of their translation in C. The columns are computed as follows: **OCamlOpt** is the time for executing programs compiled with the `ocaml-opt` compiler with the options `-unsafe -inline 50`. **Tweaked** are execution times for programs hand-optimized post-generation by the following transformations²: 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. 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)**.

Without hand-optimizing the generated code, offshoring always outperforms the native code compiler. After hand-optimizing the generated code, the situation is slightly different. In two cases, the OCaml native code compiler outperforms `gcc`. In fact, for the `ks` benchmark, the OCaml compiler does much better than `gcc`.

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.

A comparison of the sizes of binaries (Table 4) generated by the OCaml native code compiler with the offshored code compiled with `gcc` shows that compiling using `gcc` resulted in binaries of much smaller sizes. Even for the **Tweaked** sources, the binary sizes ranged from 4 to 20 times the sizes of the `gcc` generated binaries. For `icc`, there was also a saving in binary sizes. OCaml-generated binaries were between 3 to 6 times larger than those generated by `icc`. The primary reason for the larger OCaml image is that it has a bigger runtime system than C.

² Thanks to Xavier Leroy for pointing out that these changes would improve the relative performance of the OCaml native code compiler.

Name	Generated	OCamlOpt	Ratio'	Tweaked	Ratio''
forward	13103	344817	26.3x	256204	19.6x
gib	11811	279205	23.6x	262243	22.2x
knapsack	41910	270031	6.44x	264012	6.30x
lcs	59037	341085	5.78x	251541	4.26x
obst	40711	213933	5.26x	213999	5.26x
opt	45389	234512	5.17x	234492	5.17x

Using GNU's gcc

Name	Generated	OCamlOpt	Ratio	Tweaked	Ratio''
forward	43957	344817	7.84x	256204	5.83x
gib	43270	279205	6.45x	262243	6.06x
knapsack	43425	270031	6.22x	264012	6.08x
lcs	84296	341085	4.05x	251541	2.98x
obst	68029	213933	3.15x	213999	3.15x
opt	69176	234512	3.39x	234492	3.39x

Using Intel's icc

Table 4. Binary Image Sizes of OCaml Native Code Compiled vs. Offshored

6 Further Opportunities for Offshoring

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 fairly systematically. In particular, 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. Arbitrary dimension arrays are a natural extension of the two-dimensional representation that we have already addressed.

7 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 become 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 take MetaOCaml as the base language. Experimental results indicate that this approach yields significantly better performance as compared to the OCaml bytecode compiler, and often better performance than the OCaml native code compiler. We have implemented this extension in MetaOCaml. A fully automated version of our performance measurement suite has been implemented and made available online [3]. An offshoring translation targeting FORTRAN is under development.

Acknowledgments: We would like to thank John Mellor-Crummey and Gregory Malecha, who read drafts of this paper and gave us several helpful suggestions. We would also like to thank the reviewers and the editors for their detailed and constructive comments. One of reviewers asked whether offshoring can be used to produce highly-optimized libraries such as those used in the SPIN model checker or in Andrew Goldberg’s network optimization library. We thank the reviewer for the intriguing suggestion, and we hope that further experience with offshoring will allow us to answer this question.

References

1. Cristiano Calcagno, Eugenio Moggi, and Walid Taha. ML-like inference for classifiers. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.
2. Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 14 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 (extended version). Available online from: <http://www.cs.rice.edu/~taha/publications/preprints/2005-06-28-TR.pdf>, June 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 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.
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*. 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 *the International Conference on Functional Programming (ICFP '02)*, 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, and Oleg Kiselyov. Staging dynamic programming algorithms, April 2005. Unpublished manuscript, available from <http://www.cs.rice.edu/~taha/publications.html>.
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)*, Boston, 2000. ACM Press.
19. Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 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.

A Type System for the C Fragment

The type system for the target language is defined for a particular Σ that assigns types to constants and operators.

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

$$\begin{array}{c}
\frac{c : b \in \Sigma}{\Gamma \vdash c : b} \text{Const} \quad \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \text{Var} \quad \frac{f^{(1)} : b \ f(b_1) \in \Sigma}{\Gamma \vdash f^{(1)}(e) : b} \text{Op1} \quad \frac{f^{[2]} : b \ f^{[2]}(b_1, b_2) \in \Sigma}{\Gamma \vdash e_1 \ f^{[2]} \ e_2 : b} \text{Op2} \\
\\
\frac{\Gamma(x) = b \ \{t_i\}_{i \in \{\dots n\}}}{\Gamma \vdash e_i : t_i}_{i \in \{\dots n\}} \text{FCall} \quad \frac{\Gamma(x) = \text{int}}{\Gamma \vdash x++ : \text{int}} \text{Inc} \quad \frac{\Gamma(x) = \text{int}}{\Gamma \vdash x-- : \text{int}} \text{Dec} \\
\\
\frac{\Gamma(x) = b \ []}{\Gamma \vdash e : \text{int}} \text{Arr1} \quad \frac{\Gamma(x) = * b \ []}{\Gamma \vdash e_1 : \text{int}} \text{Arr2} \quad \frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash e_1 ? e_2 : e_3 : t} \text{If}
\end{array}$$

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

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:

$$r ::= t \mid \perp$$

The r types indicate return contexts: if a statement is well-formed with respect to the return context \perp , 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.

$$\begin{array}{c}
\frac{\Gamma \vdash e : t_1}{\Gamma \vdash e : \perp} \text{EStat} \quad \frac{\Gamma(x) = b}{\Gamma \vdash x = e : \perp} \text{Assign} \quad \frac{\Gamma(x) = b \ []}{\Gamma \vdash e_1 : \text{int}} \text{SetArr1} \\
\\
\frac{\Gamma(x) = * b \ [] \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 : \text{int}} \text{SetArr2} \quad \frac{\Gamma \vdash e : \text{int} \quad \Gamma \vdash s_1 : r \quad \Gamma \vdash s_2 : r}{\Gamma \vdash \text{if } (e) \ s_1 \ \text{else } s_2 : r} \text{IfS} \quad \frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash s : \perp} \text{While} \\
\\
\frac{\Gamma(x) = \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 : \text{int}} \text{For} \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \text{return } (e) : t} \text{Ret} \\
\\
\frac{\Gamma \vdash c_n : \text{int} \quad \{ \Gamma \vdash s_i : \perp \}_{i \in \{\dots n\}}}{\Gamma \vdash e : \text{int}} \text{Sw} \quad \frac{\{ \Gamma \cup (d_j)^{j \in \{\dots m\}} \vdash s_i \}_{i \in \{\dots n-1\}} : \perp}{\Gamma \cup (d_j)^{j \in \{\dots m\}} \vdash s_n : r} \text{Blk}
\end{array}$$

Function definition ($\Gamma \vdash f$)

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

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

$$\frac{}{\Gamma \vdash \cdot} \text{Empty} \quad \frac{\Gamma \vdash b \ f \ (a_i)^{i \in \{\dots n\}} \ s}{\Gamma \vdash (b \ f \ (a_i)^{i \in \{\dots n\}} \ s) ; g} \text{ExtTop}$$

Lemma 1 (Weakening).

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

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

B Type System for the OCaml Fragment

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

$$\begin{array}{c}
\frac{\hat{c} : \hat{b} \in \Sigma}{\Gamma \vdash \hat{c} : \hat{b}} \text{Const} \quad \frac{\Gamma(\hat{x}) = \hat{t}}{\Gamma \vdash \hat{x} : \hat{t}} \text{Var} \\
\\
\frac{\Gamma \vdash \hat{x} : (\hat{p}_1, \dots, \hat{p}_n) \rightarrow \hat{b} \quad \{\Gamma \vdash \hat{e}_i : \hat{p}_i\}_{i \in \{1 \dots n\}}}{\Gamma \vdash \hat{x} (\hat{e}_i)_{i \in \{1 \dots n\}} : \hat{b}} \text{FCall} \quad \frac{\hat{f}^{(1)} : \hat{b}_1 \rightarrow \hat{b} \in \Sigma \quad \Gamma \vdash \hat{e} : \hat{b}_1}{\Gamma \vdash \hat{f}^{(1)} \hat{e} : \hat{b}} \text{Fun[1]} \quad \frac{\hat{f}^{(2)} : \hat{b}_1 \rightarrow \hat{b}_2 \rightarrow \hat{b} \in \Sigma \quad \Gamma \vdash \hat{e}_1 : \hat{b}_1 \quad \Gamma \vdash \hat{e}_2 : \hat{b}_2}{\Gamma \vdash \hat{f}^{(2)} \hat{e}_1 \hat{e}_2 : \hat{b}} \text{Fun[2]} \\
\\
\frac{\hat{f}^{[2]} : \hat{b}_1 \rightarrow \hat{b}_2 \rightarrow \hat{b} \in \Sigma \quad \Gamma \vdash \hat{e}_1 : \hat{b}_1 \quad \Gamma \vdash \hat{e}_2 : \hat{b}_2}{\Gamma \vdash \hat{e}_1 \hat{f}^{[2]} \hat{e}_2 : \hat{b}} \text{Fun[2]} \quad \frac{\Gamma \vdash \hat{e}_1 : \text{bool} \quad \Gamma \vdash \hat{e}_2 : \hat{t} \quad \Gamma \vdash \hat{e}_3 : \hat{t}}{\Gamma \vdash \text{if } \hat{e}_1 \text{ then } \hat{e}_2 \text{ else } \hat{e}_3 : \hat{t}} \text{If} \\
\\
\frac{\Gamma(\hat{x}) = \hat{t} \text{ ref}}{\Gamma \vdash !\hat{x} : \hat{t}} \text{Ref} \quad \frac{\Gamma \vdash \hat{e} : \text{int} \quad \Gamma(\hat{x}) = \hat{b} \text{ array}}{\Gamma \vdash \hat{x}.(\hat{e}) : \hat{b}} \text{Arr1} \quad \frac{\Gamma \vdash \hat{e}_1 : \text{int} \quad \Gamma \vdash \hat{e}_2 : \text{int} \quad \Gamma(\hat{x}) = \hat{b} \text{ array array}}{\Gamma \vdash \hat{x}.(\hat{e}_1).(\hat{e}_2) : \hat{b}} \text{Arr2}
\end{array}$$

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

$$\begin{array}{c}
\frac{\Gamma \vdash e : t}{\Gamma \vdash e : t} \text{DExp} \quad \frac{\Gamma \vdash \hat{d}_1 : \text{unit} \quad \Gamma \vdash \hat{d}_2 : \hat{t}}{\Gamma \vdash \hat{d}_1; \hat{d}_2 : \hat{t}} \text{SeqD} \quad \frac{\Gamma \vdash \hat{e} : \hat{b} \quad \Gamma, \hat{x} : \hat{b} \vdash \hat{d} : \hat{t}}{\Gamma \vdash \left\{ \text{let } \hat{x} : \hat{b} = \hat{e} \right\} : \hat{t}} \text{Let} \quad \frac{\Gamma \vdash \hat{c} : \hat{b} \quad \Gamma, \hat{x} : \hat{b} \text{ ref} \vdash \hat{d} : \hat{t}}{\Gamma \vdash \left\{ \text{let } \hat{x} : \hat{b} \text{ ref} = \text{ref } \hat{c} \right\} : \hat{t}} \text{LetRef} \\
\\
\frac{\Gamma \vdash \hat{c}_2 : \hat{b} \quad \Gamma, \hat{x} : \hat{b} \text{ array} \vdash \hat{d} : \hat{t}}{\Gamma \vdash \left\{ \text{let } \hat{x} : \hat{b} \text{ array} = \text{Array.make } \hat{c}_1 \hat{c}_2 \text{ in } \hat{d} \right\} : \hat{t}} \text{LetArr1} \quad \frac{\Gamma \vdash \hat{c}_1 : \text{int} \quad \Gamma \vdash \hat{c}_3 : \hat{b} \quad \Gamma, \hat{x} : \hat{b} \text{ array array} \vdash \hat{d} : \hat{t}}{\Gamma \vdash \left\{ \text{let } \hat{x} : \hat{b} \text{ array array} = \text{Array.make_matrix } \hat{c}_1 \hat{c}_2 \hat{c}_3 \text{ in } \hat{d} \right\} : \hat{t}} \text{LetArr2} \quad \frac{\Gamma \vdash \hat{e} : \hat{b} \quad \Gamma(\hat{x}) = \hat{b} \text{ ref}}{\Gamma \vdash (\hat{x} := \hat{e}) : \text{unit}} \text{Ref} \\
\\
\frac{\Gamma(\hat{x}) = \hat{b} \text{ array} \quad \Gamma \vdash \hat{e}_1 : \text{int} \quad \Gamma \vdash \hat{e}_2 : \hat{b}}{\Gamma \vdash \hat{x}.(\hat{e}_1) \leftarrow (\hat{e}_2) : \text{unit}} \text{SetArr1} \quad \frac{\Gamma(\hat{x}) = \hat{b} \text{ array array} \quad \Gamma \vdash \hat{e}_2 : \text{int} \quad \Gamma \vdash \hat{e}_1 : \text{int} \quad \Gamma \vdash \hat{e}_3 : \hat{b}}{\Gamma \vdash \hat{x}.(\hat{e}_1).(\hat{e}_2) \leftarrow (\hat{e}_3) : \text{unit}} \text{SetArr2} \quad \frac{\Gamma \vdash \hat{e} : \text{bool} \quad \Gamma \vdash \hat{d}_1 : \hat{t} \quad \Gamma \vdash \hat{d}_2 : \hat{t}}{\Gamma \vdash \text{if } \hat{e} \text{ then } \hat{d}_1 \text{ else } \hat{d}_2 : \hat{t}} \text{IfD} \\
\\
\frac{\Gamma \vdash \hat{e} : \text{bool} \quad \Gamma \vdash \hat{d} : \text{unit}}{\Gamma \vdash \left\{ \text{while } \hat{e} \text{ do } \hat{d} \text{ done} \right\} : \text{unit}} \text{While} \quad \frac{z \in \{\text{to}, \text{downto}\} \quad \Gamma \vdash \hat{e}_2 : \text{int} \quad \Gamma \vdash \hat{e}_1 : \text{int} \quad \Gamma, \hat{x} : \text{int} \vdash \hat{d} : \text{unit}}{\Gamma \vdash \left\{ \text{for } \hat{x} : \text{int} = \hat{e}_1 \text{ z } \hat{e}_2 \text{ do } \hat{d} \text{ done} \right\} : \text{unit}} \text{For} \quad \frac{\hat{b} \neq \text{float} \quad \Gamma \vdash \hat{e} : \hat{b} \quad \{\Gamma \vdash \hat{d}_i : \hat{t}\}_{i \in \{1 \dots n\}} \quad \{\Gamma \vdash \hat{c}_i : \hat{b}\}_{i \in \{1 \dots n\}} \quad \Gamma \vdash \hat{d} : \text{unit}}{\Gamma \vdash \left\{ \text{match } \hat{e} \text{ with } ((\hat{c}_i \rightarrow \hat{d}_i))_{i \in \{1 \dots n\}} \mid _ \leftarrow \hat{d} \right\} : \text{unit}} \text{Match}
\end{array}$$

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

$$\begin{array}{c}
\frac{\Gamma, (\hat{x}_i : \hat{p}_i)^{i \in \{\dots n\}} \vdash \hat{d} : \hat{b}}{\Gamma \vdash \lambda (\hat{x}_i : \hat{p}_i)^{i \in \{\dots n\}} . (\hat{d} : \hat{b}) : (p_i)^{i \in \{\dots n\}} \rightarrow \hat{b}} \text{Toplevel} \quad \frac{\Gamma \vdash \hat{c} : \hat{b}}{\Gamma \vdash \left\{ \begin{array}{l} \text{let } \hat{x} : \hat{b} = \hat{c} \\ \text{in } \hat{s} \end{array} \right\} : \hat{t}} \text{LetS} \\
\\
\frac{\Gamma \vdash \hat{c} : \hat{b}}{\Gamma \vdash \left\{ \begin{array}{l} \text{let } \hat{x} : \hat{b} \text{ ref} = \text{ref } \hat{c} \\ \text{in } \hat{s} \end{array} \right\} : \hat{t}} \text{LetRefS} \quad \frac{\Gamma \cup (\hat{x}_i : \hat{p}_i)^{i \in \{m \dots n\}} \vdash \hat{d} : \hat{b}}{\Gamma \cup (\hat{x}_i : \hat{p}_i)^{i \in \{m \dots n\}}, \hat{x} : (\overline{p}_n \rightarrow \hat{b}) \vdash \hat{s} : \hat{t}} \text{LetFun} \\
\\
\frac{\Gamma \vdash \hat{c}_1 : \text{int} \quad \Gamma \vdash \hat{c}_3 : \hat{b}}{\Gamma \vdash \left\{ \begin{array}{l} \text{let } \hat{x} : \hat{b} \text{ array array} = \\ \text{Array.make_matrix } \hat{c}_1 \hat{c}_2 \hat{c}_3 \text{ in } \hat{s} \end{array} \right\} : \hat{t}} \text{LetArr2S} \quad \frac{\Gamma \vdash \hat{c}_1 : \text{int} \quad \Gamma, \hat{x} : \hat{b} \text{ array} \vdash \hat{s} : \hat{t}}{\Gamma \vdash \left\{ \begin{array}{l} \text{let } \hat{x} : \hat{b} \text{ array} = \\ \text{Array.make } \hat{c}_1 \hat{c}_2 \text{ in } \hat{s} \end{array} \right\} : \hat{t}} \text{LetArr1S}
\end{array}$$

C Offshoring Translation

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

$\llbracket \hat{t} \rrbracket$

```

[[int]] = int
[[bool]] = int
[[char]] = char
[[float]] = double
[[b array]] = [[b]] []
[[b array array]] = *[[b]] []

```

$\llbracket \hat{e} \rrbracket$

```

[[c]] = c
[[x]] = x
[[x(e1, ..., en)]] = x([[e1]], ..., [[en]])
[[f(1)e]] = [[f(1)]] [[e]]
[[f(2)e1e2]] = [[f(2)]] [[e1]] [[e2]]
[[e1f[2]e2]] = [[e1]] [[f[2]]] [[e2]]
[[if e1 then e2 else e3]] = [[e1]] ? [[e2]] : [[e3]]
[[!x]] = x
[[x.(e)]] = x[[[e]]]
[[x.(e1).(e2)]] = x[[[e1]]] [[e2]]

```

The statement subset of OCaml is translated to a pair (\bar{d}, \bar{s}) , where \bar{d} are declarations of variables that are bound in OCaml `let` expressions, and \bar{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 \perp or \top .

Return context $a ::= \perp \mid \top$

If the return context is \perp , 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.

$\{\hat{d}\}_a = (d, s)$

$$\begin{array}{c}
\overline{\{\hat{e}\}_L = (\cdot, \llbracket \hat{e} \rrbracket)} \quad \overline{\{\hat{e}\}_R = (\cdot, \text{return } \llbracket \hat{e} \rrbracket)} \quad \overline{\{\hat{x} := \hat{e}\}_a = (\cdot, x = \llbracket \hat{e} \rrbracket)} \quad \overline{\{\hat{x} . (\hat{e}_1) \leftarrow \hat{e}_2\}_a = (\cdot, x[[\hat{e}_1]] = \llbracket \hat{e}_2 \rrbracket)} \\
\overline{\{\hat{d}\}_a = (l, s)} \\
\overline{\{\hat{x} . (\hat{e}_1) . (\hat{e}_2) \leftarrow \hat{e}_3\}_a = (\cdot, x[[\hat{e}_1]] [[\hat{e}_2]] = \llbracket \hat{e}_3 \rrbracket)} \quad \overline{\{\left\{ \begin{array}{l} \text{let } \hat{x} : \hat{t} = \hat{e} \\ \text{in } \hat{d} \end{array} \right\}\}_a = ((\llbracket \hat{t} \rrbracket x; l), (x = \llbracket \hat{e} \rrbracket; s))}
\end{array}$$

$$\begin{array}{c}
\frac{\frac{\{\hat{d}_1\}_L = (l_1, s_1)}{\{\hat{d}_2\}_a = (l_2, s_2)}}{\{\hat{d}_1; \hat{d}_2\}_a = (l_1; l_2, s_1; s_2)} \quad \frac{\{\hat{d}\}_a = (l, s)}{\{\text{let } \hat{x} : \text{ref } \hat{t} = \text{ref } \hat{c} \text{ in } \hat{d}\}_a = ((\llbracket t \rrbracket)(c)x; l), (x = \llbracket c \rrbracket; s))} \\
\frac{\{\hat{d}\}_a = (l, s)}{\left\{ \left\{ \text{let } \hat{x} : \hat{t} \text{ array} = \text{Array.make } \hat{c}_1 \ \hat{c}_2 \text{ in } \hat{d} \right\} \right\}_a = \left(\llbracket t \rrbracket x[] = \{\overline{c_2}^{c_1 \text{ times}}\}, l; s \right)} \quad \frac{\{\hat{d}\}_a = (l, s) \quad \overline{y_m} \text{ fresh}}{\{\text{let } \hat{x} : \hat{t} \text{ array array} = \text{Array.make_matrix } \hat{e} \ \hat{c}_2 \ \hat{c}_3 \text{ in } \hat{d}\}_a = \left(\llbracket t \rrbracket y_m = \{\overline{[e]}^{c_2 \text{ times}}\}^{c_1 \text{ times}}; \llbracket t \rrbracket * x[] = \{y_1, \dots, y_{c_1}\}; l, s \right)} \\
\frac{\frac{\{\hat{d}_1\}_a = (l_1, s_1)}{\{\hat{d}_2\}_a = (l_2, s_2)}}{\{\text{if } (\hat{e}) \text{ then } \hat{d}_1 \text{ else } \hat{d}_2\}_a = (\cdot, \text{if } (\llbracket e \rrbracket)\{l_1; s_1\} \text{ else } \{l_2; s_2\})} \quad \frac{\{\hat{d}\}_L = (l, s)}{\{\text{while } (\hat{e}) \text{ do } \hat{d} \text{ done}\}_a = (\cdot, \text{while } (\llbracket e \rrbracket)\{l; s\})} \\
\frac{\{\hat{d}\}_L = (l, s)}{\{\text{for } \hat{x} = \hat{e}_1 \text{ to } \hat{e}_2 \text{ do } \hat{d} \text{ done}\}_a = (\text{int } x, \text{for } (x = \llbracket e_1 \rrbracket; x <= \llbracket e_2 \rrbracket; x++)\{l; s\})} \quad \frac{\{\hat{d}\}_L = (l, s)}{\{\text{for } \hat{x} = \hat{e}_1 \text{ downto } \hat{e}_2 \text{ do } \hat{d} \text{ done}\}_a = (\text{int } x, \text{for } (x = \llbracket e_1 \rrbracket; x >= \llbracket e_2 \rrbracket; x--)\{l; s\})} \\
\frac{\frac{\{\hat{d}_n\}_{L_n} = (l_n, s_n) \quad \{\hat{d}\}_L = (l, s)}{\{\text{match } \hat{e} \text{ with } \hat{c}_n \rightarrow \hat{d}_{n_n} \mid _ \rightarrow \hat{d}\}_a = (\text{case } c_n : \{l_n; s_n; \text{break}\}; \text{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.

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

$$\begin{array}{c}
\frac{\langle \hat{s} \rangle = (g, l)}{\langle \text{let } \hat{x} : \hat{t} = \hat{c} \text{ in } \hat{s} \rangle = (g, (\llbracket t \rrbracket x = c; l))} \quad \frac{\langle \hat{s} \rangle = (g, l)}{\langle \text{let } \hat{x} : \text{ref } \hat{t} = \text{ref } \hat{c} \text{ in } \hat{s} \rangle = (g, (\llbracket t \rrbracket x = c; l))} \quad \frac{\langle \hat{s} \rangle = (g, l)}{\langle \left\{ \left\{ \text{let } \hat{x} : \hat{t} \text{ array} = \text{Array.make } \hat{c}_1 \ \hat{c}_2 \text{ in } \hat{s} \right\} \right\} \rangle = (g, (\llbracket t \rrbracket x[] = \{\overline{c_2}^{c_1 \text{ times}}\}; l))} \\
\frac{\langle \hat{d} \rangle = (g, l) \quad (y_i \text{ fresh})^{i \in \{1 \dots c_1\}}}{\langle \text{let } \hat{x} : \hat{t} \text{ array array} = \text{Array.make_matrix } \hat{e} \ \hat{c}_2 \ \hat{c}_3 \text{ in } \hat{d} \rangle_a = (g, (\llbracket t \rrbracket y_i = \{\overline{[e]}^{c_2 \text{ times}}\}^{i \in \{1 \dots c_1\}}; \llbracket t \rrbracket * x[] = \{y_1, \dots, y_{c_1}\}; l))} \\
\frac{\langle \hat{s} \rangle = (g, l) \quad \langle \hat{d} \rangle_R = (l_d, s_d)}{\langle \text{let } \hat{f} (\hat{x}_i : \hat{p}_i)^{i \in \{1 \dots n\}}; \hat{b} = \hat{d} \text{ in } \hat{s} \rangle = (\llbracket b \rrbracket f (\llbracket p \rrbracket_i x_i)^{i \in \{1 \dots n\}} \{l_d; s_d\}; g, l)} \quad \frac{\langle \hat{d} \rangle_R = (l, s)}{\langle \lambda (\hat{x}_i : \hat{p}_i)^{i \in \{1 \dots n\}}. (\hat{d} : \hat{b}) \rangle = ((\llbracket b \rrbracket \text{procedure } (\llbracket p \rrbracket_i x_i)^{i \in \{1 \dots n\}} \{l; s\}), \cdot)}
\end{array}$$

Lemma 2 (Basic properties of translation).

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

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

D Type Preservation

The main technical result we present in this paper is that the offshoring translation preserves well-typedness of programs: a well-typed OCaml program from the designated subset of OCaml is translated into a well-typed C program in the designated subset of C.

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

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

Lemma 3 (Type preservation for expressions and statements).

1. $\forall \hat{\Gamma}, \hat{e}, \hat{t}. \text{ if } \hat{\Gamma} \vdash \hat{e} : \hat{\tau}, \text{ then } \llbracket \hat{\Gamma} \rrbracket \vdash \llbracket \hat{e} \rrbracket : \llbracket \hat{t} \rrbracket$
2. $\forall \hat{\Gamma}, \hat{d}. \text{ if } \hat{\Gamma} \vdash \hat{d} : \hat{t}, \text{ and } \{\hat{d}\}_L = (\bar{l}, \bar{s}), \text{ then } \llbracket \hat{\Gamma} \rrbracket \vdash \{l, s\} : \perp$
3. $\forall \hat{\Gamma}, \hat{d}. \text{ if } \hat{\Gamma} \vdash \hat{d} : \hat{t}, \text{ and } \{\hat{d}\}_R = (\bar{l}, \bar{s}), \text{ then } \llbracket \hat{\Gamma} \rrbracket \vdash \{l, s\} : \llbracket \hat{t} \rrbracket$

Theorem 1 (Type preservation). If $\hat{\Gamma} \vdash \hat{s} : \hat{a}_n \rightarrow \hat{b}$ and $\langle \hat{s} \rangle = (g, l)$, then $\llbracket \hat{\Gamma} \rrbracket \cup |l| \cup |g| \vdash g$.

Proof is by induction on the height of various derivations.

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

Assumption 1 Variable

1. For any finite sets of terms, no two bound variables have the same name.
2. For any two terms, the set of bound variables in one, and the set of free variables in the other are disjoint.

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

Proof (Theorem 1.1). Proof is by induction on the height of the typing derivation of expressions.

1. **Case** $\hat{\Gamma} \vdash \hat{c} : \hat{t}$. Constants are translated to constants, the rules are virtually identical.
2. **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 $\llbracket \cdot \rrbracket$, $\llbracket \hat{\Gamma} \rrbracket(x) = \llbracket \hat{t} \rrbracket$, and therefore $\llbracket \hat{\Gamma} \rrbracket \vdash x : \llbracket \hat{t} \rrbracket$.
3. **Case** $\hat{\Gamma} \vdash f^{(1)} \hat{e} : \hat{b}$. By inversion
 - (a) $f^{(1)} : \hat{b}_1 \rightarrow \hat{b} \in \Sigma$
 - (b) $\hat{\Gamma} \vdash \hat{e} : \hat{b}_1$.

We may apply the induction hypothesis to (3b) to obtain $\llbracket \hat{\Gamma} \rrbracket \vdash \llbracket \hat{e} \rrbracket : \llbracket \hat{b}_1 \rrbracket$. Then, by definition of \vdash , $\llbracket \hat{\Gamma} \rrbracket \vdash f^{(1)}(\llbracket \hat{e} \rrbracket) : \llbracket \hat{b} \rrbracket$.

4. **Case** $\hat{\Gamma} \vdash \hat{e}_1 f^{[2]} \hat{e}_2 : \hat{b}$. Similar to previous case.
5. **Case** $\hat{\Gamma} \vdash \hat{x} (\hat{e}_1, \dots, \hat{e}_2) : \hat{b}$. By inversion we have
 - (a) $\hat{\Gamma} \vdash \hat{x} : (\hat{p}_1, \dots, \hat{p}_n) \rightarrow \hat{b}$ thus $\hat{\Gamma}(\hat{x}) = (\hat{p}_i)^{i \in \{1 \dots n\}} \rightarrow \hat{b}$.
 - (b) $\forall i. 0 \leq i \leq n, \hat{\Gamma} \vdash \hat{e}_i : \hat{p}_i$.
 We apply the induction hypothesis to (5b) to obtain $\forall i. 0 \leq i \leq n, \llbracket \hat{\Gamma} \rrbracket \vdash \llbracket \hat{e}_i \rrbracket : \llbracket \hat{p}_i \rrbracket$. Then, by definition of \vdash , $\llbracket \hat{\Gamma} \rrbracket \vdash x(\llbracket \hat{e}_1 \rrbracket, \dots, \llbracket \hat{e}_n \rrbracket) : \llbracket \hat{b} \rrbracket$.
6. **Case** $\hat{\Gamma} \vdash \text{if } \hat{e}_1 \text{ then } \hat{e}_2 \text{ else } \hat{e}_3 : \hat{t}$. Inversion, followed by the induction hypothesis, we obtain $\llbracket \hat{\Gamma} \rrbracket \vdash \llbracket \hat{e}_1 \rrbracket : \text{int}, \llbracket \hat{\Gamma} \rrbracket \vdash \llbracket \hat{e}_2 \rrbracket : \llbracket \hat{t} \rrbracket, \llbracket \hat{\Gamma} \rrbracket \vdash \llbracket \hat{e}_3 \rrbracket : \llbracket \hat{t} \rrbracket$. Then, by definition of \vdash , $\llbracket \hat{\Gamma} \rrbracket \vdash \llbracket \hat{e}_1 \rrbracket ? \llbracket \hat{e}_2 \rrbracket : \hat{e}_3 : \llbracket \hat{t} \rrbracket$.
7. **Case** $\hat{\Gamma} \vdash !\hat{x} : \hat{t}$. Base case. By inversion, $\hat{\Gamma}(\hat{x}) = \hat{t} \text{ ref}$. By definition of $\llbracket \hat{\Gamma} \rrbracket$, $\llbracket \hat{\Gamma} \rrbracket(x) = \llbracket \hat{t} \rrbracket$. Then, easily by definition of \vdash , $\llbracket \hat{\Gamma} \rrbracket \vdash x : \llbracket \hat{t} \text{ ref} \rrbracket$.
8. **Case** $\hat{\Gamma} \vdash \hat{x}.\hat{e} : \hat{b}$. By inversion of \vdash , we obtain
 - (a) $\hat{\Gamma} \vdash \hat{e} : \text{int}$
 - (b) $\hat{\Gamma}(\hat{x}) = \hat{b} \text{ array}$
 We can apply the induction hypothesis to (8a) to obtain $\llbracket \hat{\Gamma} \rrbracket \vdash \llbracket \hat{e} \rrbracket : \text{int}$. Also, by definition of $\llbracket \hat{\Gamma} \rrbracket$, $\llbracket \hat{\Gamma} \rrbracket(x) = b[]$. Applying the **Arr1** rule of \vdash , we obtain $\llbracket \hat{\Gamma} \rrbracket \vdash \llbracket \hat{x}.\hat{e} \rrbracket : \llbracket \hat{b} \rrbracket$.
9. **Case** $\hat{\Gamma} \vdash \hat{x}.\hat{e}_1.\hat{e}_2 : \hat{b}$. Similar to (8). □

Proof (Theorem 1.2).

1. **Case** $\hat{\Gamma} \vdash \hat{e} : \hat{t}$. We know that $\hat{\Gamma} \vdash \hat{e} : \hat{t}$. By translation $\{\hat{e}\}_L = (\cdot, \llbracket \hat{e} \rrbracket)$ or $\{\hat{e}\}_R = (\cdot, \text{return } \llbracket \hat{e} \rrbracket)$. By type preservation of expressions both $\llbracket e \rrbracket$ and $\text{return } \llbracket e \rrbracket$ are well-typed.
2. **Case** $\hat{\Gamma} \vdash \hat{d}_1; \hat{d}_2 : \hat{t}$. Given that $\hat{\Gamma} \vdash \hat{d}_1; \hat{d}_2 : \hat{t}$, by inversion we know that
 - (a) $\hat{\Gamma} \vdash \hat{d}_1 : \text{unit}$
 - (b) $\hat{\Gamma} \vdash \hat{d}_2 : \hat{t}$ Let $\{\hat{d}_1\}_L = (d_1, s_1)$ and $\{\hat{d}_2\}_a = (d_2, s_2)$. By translation, $\{\hat{d}_1; \hat{d}_2\}_a = (l_1; l_2, s_1; s_2)$. We are trying to show $\llbracket \hat{\Gamma} \rrbracket \vdash (l_1, l_2, s_1, s_2) : \hat{t}$. We apply the induction hypothesis to (8a and 8b) to obtain $\llbracket \hat{\Gamma} \rrbracket \vdash (d_1, s_1) : \perp$ and $\llbracket \hat{\Gamma} \rrbracket \vdash (d_2, s_2) : \llbracket \hat{t} \rrbracket$. The desired result follows the C typing rule for $s_1; s_2$.
3. **Case** $\hat{\Gamma} \vdash \hat{x} := \hat{e} : \text{unit}$. By inversion, we know that $\hat{\Gamma} \vdash \hat{e} : \hat{b}$ and $\hat{\Gamma}(\hat{x}) = \hat{b} \text{ ref}$. Let $\{\hat{x} := \hat{e}\} = (\cdot, x = \llbracket \hat{e} \rrbracket)$. By type preservation of expressions, we know that $\llbracket \hat{\Gamma} \rrbracket \vdash e : \llbracket \hat{b} \rrbracket$. By definition of $\llbracket \hat{\Gamma} \rrbracket$, $\llbracket \hat{\Gamma} \rrbracket(x) = \llbracket \hat{b} \rrbracket$. Therefore, by C statement typing, $\llbracket \hat{\Gamma} \rrbracket \vdash^t x = \llbracket \hat{e} \rrbracket : \perp$.
4. **Case** $\hat{\Gamma} \vdash \text{let } x : \hat{b} = \hat{e} \text{ in } \hat{d} : \hat{t}$. 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 $\{\hat{d}\}_a = (l, s)$. Then we know $\{\text{let } x : \hat{b} = \hat{e} \text{ in } \hat{d}\}_a = ((\llbracket \hat{b} \rrbracket x; l), (x = \llbracket C \rrbracket; s))$. By type preservation of expressions, we know that $\llbracket \hat{\Gamma} \rrbracket \vdash e : \llbracket \hat{b} \rrbracket$. We must show that $\llbracket \hat{\Gamma} \rrbracket \vdash ((\llbracket \hat{b} \rrbracket x; l), (x = \llbracket C \rrbracket; s))$. By the induction hypothesis, $\llbracket \hat{\Gamma} \rrbracket \vdash (l, s)$. But now by definition of the C typing rule for $s_1; s_2$, the desired statement follows.
5. **Case** $\hat{\Gamma} \vdash \text{let } x : \hat{b} \text{ ref} = \text{ref } \hat{c} \text{ in } \hat{d} : \hat{t}$. By inversion of OCaml typing, we know that $\hat{\Gamma} \vdash \hat{c} : \hat{b} \text{ ref}$ and $\hat{\Gamma}, \hat{x} : \hat{b} \text{ ref} \vdash \hat{d} : \hat{t}$. Let $\{\hat{d}\}_a = (l, s)$. Then we know $\{\text{let } x : \hat{b} \text{ ref} = \text{ref } \hat{c} \text{ in } \hat{d}\}_a = ((\llbracket \hat{b} \rrbracket x; l), (x = \llbracket C \rrbracket; s))$. By type

preservation of expressions, we know that $\llbracket \hat{T} \rrbracket \vdash c : \llbracket b \rrbracket$. We must show that $\llbracket \hat{T} \rrbracket \vdash ((\llbracket b \rrbracket x; l), (x = \llbracket C \rrbracket; s))$. By the induction hypothesis, $\llbracket \hat{T} \rrbracket \vdash (d, s)$. But now by definition of the C typing rule for $s_1; s_2$, the desired statement follows.

6. **Case** $\hat{T} \vdash \text{let } x : \hat{b} \text{ array} = \text{Array.make } \hat{c}_1 \hat{c}_2 \text{ in } \hat{d} : \hat{t}$. By inversion, we know that $\hat{T} \vdash \hat{c}_1 : \text{int}$, $\hat{T} \vdash \hat{c}_2 : \hat{b}$ and that $\hat{T}, \hat{x} : \hat{b} \text{ array} \vdash \hat{d} : \hat{t}$. By translation, $\{\hat{T} \vdash \text{let } x : \hat{b} \text{ array} = \text{Array.make } \hat{c}_1 \hat{c}_2 \text{ in } \hat{d}\}_a = (\llbracket \hat{t} \rrbracket x \square = \{\overline{\hat{c}_2}^{\text{times}}\}, l; s)$ where $\{\hat{d}\}_a = (l, s)$. By type preservation of expressions, we know that $\llbracket \hat{T} \rrbracket \vdash c_1 : \text{int}$ and $\llbracket \hat{T} \rrbracket \vdash c_2 : \llbracket \hat{b} \rrbracket$. Thus we know the definition of the array is valid. Also, by definition of $\llbracket \hat{T} \rrbracket$, $\llbracket \hat{T} \rrbracket(x) = \llbracket \hat{b} \rrbracket$. We now apply the induction hypothesis to $\hat{T}, \hat{x} : \hat{b} \text{ array} \vdash \hat{d} : \hat{t}$, weakening, and the C typing rule for $s_1; s_2$ to obtain $\llbracket \hat{T} \rrbracket, b x \square \vdash \llbracket \hat{t} \rrbracket x \square = \{\overline{\hat{c}_2}^{\text{times}}\}, l; s$.
7. **Case** $\hat{T} \vdash \text{let } x : \hat{b} \text{ array array} = \text{Array.make_matrix } \hat{c}_1 \hat{c}_2 \hat{c}_3 \text{ in } \hat{d} : \hat{t}$. Similar to previous case.
8. **Case** $\hat{T} \vdash \hat{x}.(\hat{e}_1) \leftarrow \hat{e}_2 : \text{unit}$. By inversion we know:
 - (a) $\hat{T}(x) = \hat{b} \text{ array}$
 - (b) $\hat{T} \vdash \hat{e}_1 : \text{int}$
 - (c) $\hat{T} \vdash \hat{e}_2 : \hat{b}$
Let $\{\hat{T} \vdash \hat{x}.(\hat{e}_1) \leftarrow \hat{e}_2\}_a = (\cdot, x[\llbracket \hat{e}_1 \rrbracket] = \llbracket \hat{e}_2 \rrbracket)$. We apply the definition of $\{\hat{T}\}$ to 8a and type preservation of expressions to 8b and 8c to obtain
 - (a) $\{\hat{T}\}(x) = b$
 - (b) $\{\hat{T}\} \vdash \llbracket \hat{e}_1 \rrbracket : \text{int}$
 - (c) $\{\hat{T}\} \vdash \llbracket \hat{e}_2 \rrbracket : b$
By C typing for array assignment (**SetArr1**, we now have: $\{\Gamma\} \vdash x[\llbracket \hat{e}_1 \rrbracket] = \llbracket \hat{e}_2 \rrbracket : \perp$.
9. **Case** $\hat{T} \vdash \hat{x}.(\hat{e}_1).(\hat{e}_2) \leftarrow \hat{e}_3 : \text{unit}$. Similar to previous case.
10. **Case** $\hat{T} \vdash \text{if } \hat{e}_1 \text{ then } \hat{d}_1 \text{ else } \hat{d}_2 : \hat{t}$. By inversion, we know that $\hat{T} \vdash \hat{e} : \text{bool}$, $\hat{T} \vdash \hat{d}_1 : \hat{t}$ and $\hat{T} \vdash \hat{d}_2 : \hat{t}$. By translation, $\{\hat{T} \vdash \text{if } \hat{e}_1 \text{ then } \hat{d}_1 \text{ else } \hat{d}_2\}_a = (\cdot, \text{if } (\llbracket \hat{e} \rrbracket)\{l_1; s_1\} \text{ else } \{l_2; s_2\})$ where $\{\hat{d}_1\}_a = (l_1; s_1)$ and $\{\hat{d}_2\}_a = (l_2; s_2)$. From preservation of typing for expressions $\llbracket \hat{T} \rrbracket \vdash \llbracket \hat{e} \rrbracket : \text{int}$. By induction, $\llbracket \hat{T} \rrbracket \vdash (l_1, s_1) : \llbracket \hat{t} \rrbracket$ and $\llbracket \hat{T} \rrbracket \vdash (l_2, s_2) : \llbracket \hat{t} \rrbracket$. To finalize this proof we use the C **Ifs** rule.
11. **Case** $\hat{T} \vdash \text{while } \hat{e} \text{ do } \hat{d} \text{ done} : \text{unit}$. By inversion we know that $\hat{T} \vdash \hat{e} : \text{bool}$ and $\hat{T} \vdash \hat{d} : \hat{t}$. By translation, $\{\hat{T} \vdash \text{while } \hat{e} \text{ do } \hat{d} \text{ done}\}_a = (\cdot, \text{while } (\llbracket \hat{e} \rrbracket)\{l; s\})$ where $\{\hat{d}\}^L = (l; s)$. From preservation of typing for expressions $\llbracket \hat{T} \rrbracket \vdash \llbracket \hat{e} \rrbracket : \text{int}$. By induction, $\llbracket \hat{T} \rrbracket \vdash (l; s) : \perp$. To finalize this proof we use the C **While** rule.
12. **Case** $\hat{T} \vdash \text{for } \hat{x} = \hat{e}_1 \{ \text{to, downto} \} \hat{e}_2 \text{ do } \hat{d} \text{ done} : \text{unit}$. By inversion, we know that $\hat{T} \vdash \hat{e}_1 : \text{int}$, $\hat{T} \vdash \hat{e}_2 : \text{int}$ and $\hat{T}, \hat{x} : \text{int} \vdash \hat{d} : \hat{t}$. By translation, $\{\hat{T} \vdash \text{for } \hat{x} = \hat{e}_1 \text{ to } \hat{e}_2 \text{ do } \hat{d} \text{ done}\} = (\text{int } x, \text{for } (x = \llbracket \hat{e}_1 \rrbracket; x \leq \llbracket \hat{e}_2 \rrbracket; x++)\{l; s\})$ where $\{\hat{d}\} = (l; s)$. From preservation of typing for expressions $\llbracket \hat{T} \rrbracket \vdash \llbracket \hat{e}_1 \rrbracket : \text{int}$ and $\llbracket \hat{T} \rrbracket \vdash \llbracket \hat{e}_2 \rrbracket : \text{int}$. Also, by definition of $\llbracket \hat{T} \rrbracket$, $\llbracket \hat{T} \rrbracket(x) = \text{int}$. By induction, $\llbracket \hat{T} \rrbracket \vdash (l, s) : \perp$. To finalize this proof we use the C **For** rule.

13. **Case match \hat{e} with $(\hat{c}_i \rightarrow \hat{d}_i)^{i \in \{...n\}} \mid _ \leftarrow \hat{d}$: **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 $\hat{\Gamma} \vdash \hat{d} : \mathbf{unit}$. By translation, $\{\text{match } \hat{e} \text{ with } (\hat{c}_i \rightarrow \hat{d}_i)^{i \in \{...n\}} \mid _ \leftarrow \hat{d}\}_a = \overline{\text{switch}} (\llbracket e \rrbracket) \{\text{case } c_n : \{l_n; s_n; \text{break}\}; \text{default} : \{l; s\}\}$ where $\{\hat{d}_n\}^L = \overline{(l_n, s_n)}$ and $\{\hat{d}\}^L = (l, s)$. From preservation of typing for expressions $\llbracket \hat{\Gamma} \rrbracket \vdash e : \llbracket \hat{b} \rrbracket$ and $\llbracket \hat{\Gamma} \rrbracket \vdash \overline{c_n} : \mathbf{int}$. By induction, $\llbracket \hat{\Gamma} \rrbracket \vdash \overline{(l_n, s_n)} : \perp$ and $\llbracket \hat{\Gamma} \rrbracket \vdash (l, s) : \perp$. To finalize this proof we use the **C Switch** rule.

Proof (Theorem 1.3).

1. **Case $\Gamma \vdash \lambda(\hat{x}_i : \hat{p}_i)^{i \in \{...n\}} . (\hat{d} : \hat{b}) : (p_i)^{i \in \{...n\}} \rightarrow \hat{b}$.** By inversion of typing for programs, $\Gamma, (\hat{x}_i : \hat{p}_i)^{i \in \{...n\}} \vdash \hat{d} : \hat{b}$. By translation for programs, $\langle \lambda(\hat{x}_i : \hat{p}_i)^{i \in \{...n\}} . (\hat{d} : \hat{b}) : (p_i)^{i \in \{...n\}} \rightarrow \hat{b} \rangle = (\llbracket \hat{b} \rrbracket \text{ procedure } (\llbracket \hat{p}_i \rrbracket x_i)^{i \in \{...n\}} \{l; s\}, \cdot)$ where $\{\hat{d}\}^R = (l, s)$. We must show that $\llbracket \hat{\Gamma} \rrbracket \vdash \llbracket \hat{b} \rrbracket \text{ procedure } (\llbracket \hat{p}_i \rrbracket x_i)^{i \in \{...n\}} \{l; s\}$. This is true if $\llbracket \hat{\Gamma} \rrbracket, \llbracket \hat{b} \rrbracket \text{ procedure } (\llbracket a_n \rrbracket) \vdash (l, s) : \hat{b}$. following the **C TFun** typing rule, and we know this immediately by type preservation of statements.
2. **Case $\hat{\Gamma} \vdash \text{let } \hat{x} : \hat{b} = \hat{c} \text{ in } \hat{s} : \hat{t}$.** 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 $\langle \text{let } x : \hat{b} = \hat{c} \text{ in } \hat{s} \rangle = (g, (\llbracket \hat{b} \rrbracket x = c; l))$. where $\langle \hat{s} \rangle = (g, l)$. We must show that $\llbracket \hat{\Gamma} \rrbracket \cup l \cup \{b x\} \vdash g$. By the induction hypothesis, $\llbracket \hat{\Gamma}, b x \rrbracket \cup l \vdash g$. But now by definition of $\llbracket \hat{\Gamma} \rrbracket$, $\llbracket \hat{\Gamma} \rrbracket \cup l \cup \{b x\} \vdash g$.
3. **Case $\text{let } \hat{x} (x_i : \hat{p}_i)^{i \in \{...n\}} : \hat{b} = \hat{d} \text{ in } \hat{s} : \hat{t}$.** By inversion of OCaml function typing, $\Gamma \cup (\hat{x}_i : \hat{p}_i)^{i \in \{m...n\}} \vdash \hat{d} : \hat{b}$ and $\Gamma \cup (\hat{x}_i : \hat{p}_i)^{i \in \{m...n\}}, \hat{x} : (\overline{p_n} \rightarrow \hat{b}) \vdash \hat{s} : \hat{t}$. By translation we know that $\langle \text{let } \hat{x} (x_i : \hat{p}_i)^{i \in \{...n\}} : \hat{b} = \hat{d} \text{ in } \hat{s} \rangle = (\llbracket \hat{b} \rrbracket x (\llbracket p \rrbracket_i x_i)^{i \in \{...n\}} \{l_d; s_d; \}; g, l)$ where $\langle \hat{s} \rangle = (g, l)$ and $\{\hat{d}\}^R = (l_d, s_d)$. We are trying to show $\llbracket \hat{\Gamma} \rrbracket \cup l \cup \llbracket b \rrbracket x (\llbracket p_n \rrbracket) \cup g \vdash (\llbracket \hat{b} \rrbracket x (\llbracket p \rrbracket_i x_i)^{i \in \{...n\}} \{l_d; s_d; \}; g)$. By induction and weakening, $\llbracket \hat{\Gamma} \rrbracket \cup \llbracket b \rrbracket x (\llbracket p_n \rrbracket) \cup l \cup g \vdash g$. Also by preservation of statements and weakening, the function definition is well-typed, i.e.: $\llbracket \hat{\Gamma} \rrbracket \cup l \cup g \vdash (\llbracket \hat{b} \rrbracket x (\llbracket p \rrbracket_i x_i)^{i \in \{...n\}} \{l_d; s_d; \})$. By **C TFun**, $\llbracket \hat{\Gamma} \rrbracket \vdash (\llbracket \hat{b} \rrbracket x (\llbracket p \rrbracket_i x_i)^{i \in \{...n\}} \{l_d; s_d; \})$. Now by the **C program typing** $\llbracket \hat{\Gamma} \rrbracket \cup l \cup \llbracket b \rrbracket f(\llbracket a_n \rrbracket) \cup g \vdash \llbracket b \rrbracket f[\hat{x}_n](l', s'); g$.
4. **Case $\hat{\Gamma} \vdash \text{let } x : \hat{b} \text{ ref} = \text{ref } \hat{c} \text{ in } \hat{s} : \hat{t}$.** By inversion of OCaml typing, we know that $\hat{\Gamma} \vdash \hat{c} : \hat{b}$ and that $\hat{\Gamma}, \hat{x} : \hat{b} \text{ ref} \vdash \hat{s} : \hat{t}$. Let $\langle \hat{s} \rangle = (g, l)$. By translation, $\langle \text{ref} = \text{ref } \hat{c} \text{ in } \hat{s} \rangle = (g, (\{\llbracket \hat{b} \rrbracket x\} = c, l))$. We must show that $\llbracket \hat{\Gamma} \rrbracket \cup l \cup \{b x\} \vdash g$. By the induction hypothesis, $\llbracket \hat{\Gamma}, b x \rrbracket \cup l \vdash g$. But now by definition of $\llbracket \hat{\Gamma} \rrbracket$, $\llbracket \hat{\Gamma} \rrbracket \cup l \cup \{b x\} \vdash g$.
5. **Case $\hat{\Gamma} \vdash \text{let } x : \hat{b} \text{ array} = \text{Array.make } \hat{c}_1 \hat{c}_2 \text{ in } \hat{s} : \hat{t}$.** By inversion of OCaml typing, we know that $\hat{\Gamma} \vdash \hat{c}_1 : \mathbf{int}$, $\hat{\Gamma} \vdash \hat{c}_2 : \hat{b}$ and that $\hat{\Gamma}, \hat{x} : \hat{b} \text{ array} \vdash \hat{s} : \hat{t}$. Let $\langle \hat{s} \rangle = (g, l)$. By translation, $\langle \hat{\Gamma} \vdash \text{let } x : \hat{b} \text{ array} =$

$\text{Array.make } \hat{c}_1 \ \hat{c}_2 \ \text{in } \hat{s} \rangle = (g, \{\llbracket \hat{b} \rrbracket \ x \rrbracket = \{\llbracket \hat{c}_1 \rrbracket, \dots, \llbracket \hat{c}_2 \rrbracket\}\}; l)$. By type preservation of expressions, we know that $\llbracket \hat{T} \rrbracket \vdash c_1 : \text{int}$ and $\llbracket \hat{T} \rrbracket \vdash c_2 : \llbracket \hat{b} \rrbracket$. Thus we know that the definition of the array is legal. We must show that $\llbracket \hat{T} \rrbracket \cup l \cup \{b \ x \rrbracket\} \vdash g$. By the induction hypothesis, $\llbracket \hat{T}, b \ x \rrbracket \cup l \vdash g$. But now by definition of $\llbracket \hat{T} \rrbracket$, $\llbracket \hat{T} \rrbracket \cup l \cup \{b \ x \rrbracket\} \vdash g$.

6. **Case** $\hat{T} \vdash \text{let } x : \hat{b} \ \text{array array} = \text{Array.make_matrix } \hat{c}_1 \ \hat{c}_2 \ \hat{c}_3 \ \text{in } \hat{s} : \hat{t}$.
Similar to previous case.