Formal Design Environments

Brian Aydemir, Adam Granicz, and Jason Hickey*

California Institute of Technology, Pasadena CA 91125, USA,
{emre,granicz,jyh}@cs.caltech.edu,
http://mojave.cs.caltech.edu/

Abstract. We present the design of a formal integrated design environment. The long-term goal of this effort is to allow seamless interaction between software production tools and formal design and analysis tools, especially between compilers and higher-order theorem provers. The work in this report is the initial design and architecture for integration of 1) the MetaPRL logical framework, 2) a multi-language compiler we call Mojave, and 3) a generic extensible parser we call Phobos. The integration is currently performed at the level of the Mojave functional intermediate representation, allowing the use of the theorem prover for program analysis, transformation, and optimization.

1 Introduction

We are developing formal integrated design environments (FIDEs) where formal and informal tools are used in a symbiotic relationship. That is, interactions between the formal and informal parts of the FIDE are bidirectional and interdependent.

Most, if not all, existing formal design environments do not allow bidirectional interactions, especially between the theorem prover and the compiler. Yet, the system would clearly benefit from closer interaction. For example, the compiler might be able to use the theorem prover for optimization, proof validation, or program transformation. The theorem prover would benefit from the ability to formalize its own code, especially tactics.

The larger need is for effective formal programming languages. By "effective" we mean that the languages should be general enough and efficient enough to use in software production. By "formal" we mean that programs can be both specified and verified. The compiler is responsible for efficiency, the prover for formality. In order to achieve both properties simultaneously, we argue that the theorem prover and compiler must interact closely (or, equivalently, one must be folded into the other).

In this paper, we describe our initial work integrating the MetaPRL logical framework with our Mojave multi-language compiler. There are several parts that are needed for integration: 1) the compiler and theorem prover must share

* This work was supported by the ONR, grant N00014-01-1-0765; DARPA, grant F33615-98-C3613; and AFOSR, grant F49620-01-1-0361.

a common language, 2) the compiler must allow for an extended program syntax that includes specification, and 3) the compiler and prover must also agree on a common program semantics, especially operational semantics. We present the following results:

- an architecture and implementation for the MetaPRL/Mojave formal design environment,
- a shared typed intermediate language, with semantics defined in the MetaPRL implementation of the NuPRL type theory,
- an extensible front-end, called Phobos, that uses the MetaPRL rewriting system for extending and defining programming languages,
- and examples of using the theorem prover for optimization.

Section 2 describes related work. Section 3 describes the MetaPRL, Mojave, and Phobos systems individually, and Section 4 presents the combined architecture. We give example applications in Section 5, and finish with a summary of future work.

2 Related Work

This work initially started with the development of the MetaPRL system [6–8]. MetaPRL is a logical framework, designed to allow *relations* between logics. MetaPRL is also designed as a "Logical Programming Environment" (LPE) where programs, type systems, proofs, and specifications can all be defined and related to one another.

One of the problems with the MetaPRL design is that it is a layered architecture. The theorem prover is layered above the OCaml compiler [16], and the connection is unidirectional. Any task (such as parsing, type inference, and scoping) that is assigned to the compiler is not available to the formal system, hindering effective formal software development.

In another related effort, we used the NuPRL system to optimize communication protocols for the Ensemble group communication system [12, 11]. Again, this project separated the prover from the compiler. To optimize a protocol, a parser would convert the protocol and requirements into an expression in the NuPRL type theory; the prover would apply optimization tactics to generate a "fast-path;" and the result would be printed as a ML file to be compiled by the OCaml compiler. While successful, this was awkward. Furthermore, optimization strategies were defined in NuPRL, not as part of the program code, making it difficult to synchronize the formal system with new Ensemble code releases. The architecture we propose in this paper is an effort to design a system where formal properties are "first-class" program properties, and the prover/compiler interaction is seamless.

In other related areas, Sannella and Tarlecki's Extended ML [9, 10] allows mixing of program implementation and formal specification for SML programs. The ACL2 system [3] allows extensive mixing of formal specification and Common Lisp programs. Nearly all other formal systems, including systems like

HOL [5], PVS [4] and Isabelle [14], allow extensive reasoning about programs, but reasoning is usually done offline in the same manner as NuPRL.

3 Architectural Components

There are three major parts to our architecture. The MetaPRL system provides reasoning, the Mojave system provides compilation, and the Phobos system provides generic, extensible parsing. The overall system architecture is shown in Figure 1.

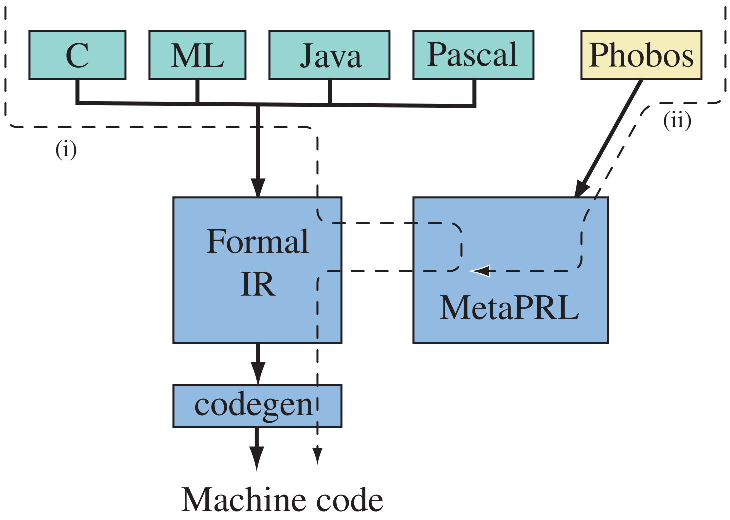

Fig. 1. System architecture: Path (i) corresponds to a traditional compilation path augmented by the theorem prover. Path (ii) uses the dynamically extensible front end.

3.1 The MetaPRL system

MetaPRL is a *logical framework*, designed along the same architectural principles as the NuPRL system. The system architecture is shown in Figure 2. The refiner contains three parts: 1) a term module, which defines the system syntax, 2) a logic engine, for theorem proving, and 3) a bookeeping module to manage and validate proofs, and perform program extraction for constructive proofs.

The meta-language defines the language of *tactics* and *conversionals* (rewriting tactics), which are used define decision procedures and proof heuristics. The entire MetaPRL system is written in OCaml, and OCaml is used as the language for tactics and conversionals.

The topmost layer in MetaPRL is the definition of theories and logics. A *theory* is defined by 1) its syntax (defined using terms), 2) its proof and rewriting

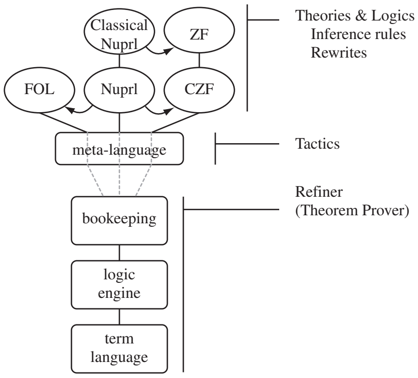

Fig. 2. MetaPRL system architecture

rules, 3) its tactics and conversionals, 4) its theorems, expressed as derived inference and rewriting rules, and 5) other utilities for display and pretty-printing.

Judgments Inference rules are often, though not always, described in a sequent logic. For example, the following inference rule would describe the implies introduction rule in a propositional logic.

$$\begin{array}{lll} \textbf{rule}\ \text{imp\_intro}\ H : & & \\ & [\text{main}] & (H, v\colon A \vdash B) \longrightarrow \\ & [\text{wf}] & (H \vdash A\ \mathit{type}) \longrightarrow \\ & & H \vdash A \Rightarrow B \end{array}$$

In this rule, the variables A, B, are meta-variables that represent arbitrary terms and H represents a context. The sequents labeled "main" and "wf" are the premises of the rule; "main" is the main premise, and "wf" is a well-formedness requirement. The $\longrightarrow$ operator is the meta-implication that MetaPRL uses to represent inference rules. The declaration of the `imp_intro` rule defines a tactic, called `imp_intro` that can used to "refine" any goal of the form $H \vdash A \Rightarrow B$ into two subgoals. From here, it is straightforward, for example, to define a derived rule (a *theorem*) that would apply to sequents of the form $H \vdash A \Rightarrow B \Rightarrow C$. The proof would use the `imp_intro` rule twice, and there would be four premises.

Rewriting judgments are defined in a similar way. The rule for beta-reduction in the untyped lambda calculus would be expressed using the following rule.

$$\textbf{rewrite}\ \text{beta} : \mathit{apply}\{\mathit{lambda}\{v.e_1[v]\}; e_2\} \longleftrightarrow e_1[e_2]$$

This declaration defines a *conversion* called `beta` that can be applied within a proof to any redex, performing the substitution. Note that the statement of the rewrite uses second-order substitution [1, 13]. The pattern $e_1[v]$ represents a term in which the variable v is allowed to be free, and the term $e_1[e_2]$ represents e_1 with e_2 substituted for v.

Syntax, and terms All logical terms, including goals and subgoals, are expressed in the language of *terms*. The general syntax of all terms has three parts. Each term has 1) an operator-name (like "sum"), which is a unique name indicating the logic and component of a term; 2) a list of parameters representing constant values; and 3) a list of subterms with possible variable bindings. We use the following syntax to describe terms, based on the NuPRL definition [2]:

$$\underbrace{opname}_{operator\ name}\ \underbrace{[p_1; \cdots; p_n]}_{parameters}\underbrace{\{\boldsymbol{v_1}.t_1; \cdots; \boldsymbol{v_m}.t_m\}}_{subterms}$$

A few examples are shown at the right. Variables are terms with a string parameter for their name; numbers have an integer parameter. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b.

Displayed form	Term
1	`number[1]{}`
$\lambda x.b$	`lambda[]{x. b}`
$f(a)$	`apply[]{f; a}`
v	`variable["v"]{}`
$x+y$	`sum[]{x; y}`

3.2 The Mojave compiler

The Mojave multi-language compiler, shown in the left half of Figure 1, is made up of three major parts. The front-ends are responsible for compiling code to the functional intermediate representation (FIR), and the back-end is responsible for generating machine code from FIR. FIR type inference and optimizations form the middle stage of the compiler. The FIR is the primary concern for this paper; it is the language we are using for interaction with MetaPRL.

Functional Intermediate Representation The FIR is designed to be a minimal, but general-purpose typed intermediate language. The FIR has higher-order functions, polymorphism, and object types. We will describe it in two parts, first the type system, and then the programs.

The FIR type system The FIR type system is based on the polymorphic lambda calculus. The type system is shown in Figure 3. There are the normal scalars, including native integers with various bit precisions ($\mathbb{Z}_8, \ldots, \mathbb{Z}_{64}$) as well as "boxed" integers, enumerations $\{0 \ldots i\}$ whose values range from 0 to $i-1$, and floating-point numbers. Enumerations are used to code several base types: the empty type *Void* is $\{0 \ldots 0\}$, *Unit* is $\{0 \ldots 1\}$, etc.

Functions have multiple arguments. The type $(t_1, \ldots, t_n) \rightarrow t$ is the space of functions taking arguments with types $t_1, \ldots, t_n$ and returning a value of type t.

Tuples $\langle t_1, \ldots, t_n \rangle$ are collections of elements having potentially different type. The t **array** represents variable-sized arrays of elements all having type t. The **data** type is used specifically for C: it represents a variable-sized data area containing elements of any type. Values of **data** type are not statically

	Type	Description
$t ::=$	$\mathbf{boxed}(\mathbb{Z})$	Boxed Integers
$\mid$	$\{0 \dots i\}$	Integer enumerations
$\mid$	$\mathbb{Z}_8, \mathbb{Z}_{16}, \mathbb{Z}_{32}, \mathbb{Z}_{64}$	Native integers
$\mid$	$\mathbf{float}$	Floating-point numbers
$\mid$	$(t_1, \dots, t_n) \rightarrow t$	Function type
$\mid$	$\mathbf{const}(v[i], t_1, \dots, t_n)$	Constructor type
$\mid$	$\langle t_1, \dots, t_n \rangle$	Tuple type
$\mid$	t $\mathbf{array}$	Array type
$\mid$	$\mathbf{data}$	Unsafe data
$\mid$	$\alpha, \beta, \dots$	Polymorphic type variables
$\mid$	v	Type variables
$\mid$	$v[t_1, \dots, t_n]$	Type application
$\mid$	$\exists \alpha.t$	Existential types
$\mid$	$\forall \alpha.t$	Universal types
$\mid$	$v.i$	Abstract type
$\mid$	$\mathbf{Object}(v.t)$	Object type
$d ::=$	$\Lambda\alpha_1, \dots, \alpha_n.\boldsymbol{t_1} + \dots + \boldsymbol{t_m}$	Type abstraction
$\gamma ::=$	$v = d \mid \alpha$	
$\Gamma ::=$	$\gamma_1, \dots, \gamma_n$	Type contexts

Fig. 3. The FIR type system

type-checked: it is not a type error to store an integer in a **data** area, and immediately fetch a value from the same location with some other type, but the runtime will raise an exception if the operation is unsafe.

Types are always defined relative to a context Γ that contains type definitions and scope information for polymorphic variables. The union type $\Lambda\alpha_1, \dots, \alpha_n.\boldsymbol{t_1} + \dots + \boldsymbol{t_m}$ is a polymorphic disjoint union of tuples $\boldsymbol{t_1}, \dots, \boldsymbol{t_m}$. A value with a disjoint union type is a tagged tuple of type $\boldsymbol{t_i}$ with tag i for some $i \in \{1, \dots, n\}$. If v defines a union type $\Lambda\alpha_1, \dots, \alpha_n.\boldsymbol{t_1} + \dots + \boldsymbol{t_n}$, then the constructor type $\mathbf{const}(v[i], a_1, \dots, a_n)$ denotes a tagged tuple of type $t_i[a_1, \dots, a_n]$.

Polymorphism is expressed using the existential and universal types. A value of type $\exists \alpha.t$ has type t for some type α, and a value of type $\forall \alpha.t$ has type t for all types α.

An object type $\mathbf{Object}(v.t)$ is a recursive type definition denoting objects with description t.

FIR expressions The FIR expressions are in a mostly-functional intermediate form where the order of evaluation is explicit. *Atoms* are values that are either constants or variables, and the other expressions are computed over the atoms.

	Definition	Description
a	$::= \mathbf{nil}(t)$	The "nil" value for type t
	$\mid\ \mathbf{boxed}(i)$	Boxed integer constants
	$\mid\ i$	Native integer constants
	$\mid\ x$	Floating-point constants
	$\mid\ v$	Variables
unop	$::= - \mid\ ! \mid \cdots$	Unary operations
binop	$::= + \mid - \mid * \mid / \mid \cdots$	Binary operations
s	$::= [i_1^1, i_2^1], \ldots, [i_1^n, i_2^n]$	Integer interval set
c	$::= s \to e$	Match case
p	$::= \mathit{bounds}(v[a_1 \ldots a_2])$	Bounds check
	$\mid\ \mathit{is\text{-}pointer}(v[a])$	Pointer check
	$\mid\ \mathit{is\text{-}function}(v[a])$	Function check
alloc	$::= \langle a_1, \ldots, a_n \rangle : t$	Tuple allocation
	$\mid\ \mathbf{const}(v[a_1, \ldots, a_n], i) \colon t$	Constructor allocation
	$\mid\ \mathit{array}(a_1, a_2, :)t$	Array allocation
	$\mid\ \mathit{malloc}(a)$	Malloc
e	$::= \mathbf{let}\, v \colon t = \mathit{unop}\, a\ \mathbf{in}\ e$	Unary operations
	$\mid\ \mathbf{let}\, v \colon t = a_1\ \mathit{binop}\ a_2\ \mathbf{in}\ e$	binary operations
	$\mid\ \mathbf{let}\, v \colon t = (\mathbf{ext}\ \texttt{"s"} : t)(a_1, \ldots, a_n)\ \mathbf{in}\ e$	Calls to the runtime
	$\mid\ v(a_1, \ldots, a_n)$	Tail call
	$\mid\ \mathbf{match}\, a\, \mathbf{with}\, c_1 \mid \cdots \mid c_n$	Set membership
	$\mid\ \mathbf{let}\, v = \mathit{alloc}\ \mathbf{in}\ e$	Allocation
	$\mid\ \mathbf{let}\, v \colon t = a_1[a_2]\ \mathbf{in}\ e$	Load
	$\mid\ a_1[a_2] \colon t \leftarrow a_3; e$	Store
	$\mid\ \mathit{assert}(p); e$	Assertion
σ	$::= v \colon t$	Variable type
	$\mid\ f \colon t = (v_1, \ldots, v_n) \to e$	Function definition
Σ	$::= \sigma_1, \ldots, \sigma_n$	Variable environment

Fig. 4. FIR expressions

Function definitions are stored in an environment Σ that also serves as a type assignment. The definitions are shown in Figure 4.

Expressions include explicit coercions and arithmetic as unary and binary operators.

The "ext" call represents a call to the runtime, providing a method to issue system calls. Type definitions for system calls are provided as part of the compiler configuration, to ensure type safety. The tailcall provides the only other means for calling a function.

The "match" construction allows pattern matching against a constant set of numbers, represented as a list of intervals. Each match case defines a set s and an expression e. Operationally, the pattern match selects the first case (s_i, e_i) such that $a \in s_i$, and evaluates the expression e_i. An inexhaustive match is a type error.

The "alloc" operation is used for allocation of tuples, constructors, arrays, or **data** arrays.

The array operations define primitives to load and store values in arrays. The store operation is the only non-functional primitive in the language.

The "assert" statement asserts that a predicate must be true. The runtime uses these predicates to validate array bounds, and other runtime properties.

3.3 Phobos

The Phobos parser provides dynamic and extensible parsing. Languages can be augmented with new syntax and semantics, and added to the system runtime dynamically.

The central issue in an extensible parser is the representation of semantic actions—the programs that describe, for each clause in the grammar, how to form the abstract syntax tree. Our approach is to represent all intermediate forms as terms, and to use the MetaPRL term rewriter to define semantic actions.

For example, Figure 5 shows the language description of simple arithmetic expressions including factorials. The entire description is represented as a language module, which can be incrementally refined and extended in inheriting modules. Based on this language module, Phobos can lex and parse a source string and return a MetaPRL term that encodes its semantic meaning.

A Phobos language module consists of

- *Term declarations*: importing terms from existing MetaPRL modules. In the above example, the arithmetic meta operations are imported from `Base_meta`, a standard MetaPRL module that defines basic operations on numbers and simple conversions for their simplification. Term declarations serve the purpose of verification and proper scoping within MetaPRL modules. Terms do not have to be declared if they are explicitly named with their parent module, for example `Itt_int_base!number`.
- *Lexical information*: terminal symbols are named and defined by their corresponding regular expressions. Upon successfully matching a regular expression, the resulting token is represented as `__token__[p:s]{'pos}`, where `p` stores the matched string, and `'pos` its source position. This term can be given further meaning by an optional lexical rewrite. In the example, numbers are rewritten to MetaPRL number terms.
- *Precedence rules*: used to define precedence and associativity of terminal symbols and production rules.
- *Grammar*: Expressed in BNF, each production may contain a list of rewrites that define the corresponding semantic action. If more than one rewrite is given, the first matching rewrite is carried out during parsing. If no rewrite is

```
Module Calculator

Terms -extend "Base_meta" {
   declare meta_sum{'e1; 'e2}, meta_diff{'e1; 'e2}
   declare meta_prod{'e1; 'e2}, meta_quot{'e1; 'e2}
}

Terms -extend "@" {
   declare fact{'e}
}

Tokens -longest {
   NUM = "[1-9][0-9]*"      { __token__[p:s]{'pos} -> Itt_int_base!number[p:n] }

   TIMES = "*"              {}
   DIV = "/"                {}
   PLUS = "+"               {}
   MINUS = "-"              {}

   LPAREN = "("             {}
   RPAREN = ")"             {}
   BANG = "!"               {}

   * EOL = "\\n"            {}
   * SPACE = " "            {}
}

%left PLUS MINUS
%left TIMES DIV
%left LPAREN RPAREN
%left BANG

Grammar -start e {
   e ::= NUM                {}
      | e PLUS e            { 'e1 PLUS 'e2     -> meta_sum{'e1; 'e2} }
      | e MINUS e           { 'e1 MINUS 'e2    -> meta_diff{'e1; 'e2} }
      | e TIMES e           { 'e1 TIMES 'e2    -> meta_prod{'e1; 'e2} }
      | e DIV e             { 'e1 DIV 'e2      -> meta_quot{'e1; 'e2} }
      | e BANG              { 'e BANG          -> fact{'e} }
      | LPAREN e RPAREN     { LPAREN 'e RPAREN -> 'e }
}

Rewrites {
   fact{1} -> 1
   fact{'number} -> meta_prod{'number; fact{meta_diff{'number; 1}}}
}
```

Fig. 5. A grammar for simple arithmetic with factorials.

given, a default rewrite is used that builds a tuple term from the right-hand side.
- *Post-parsing rewrites*: Possibly multiple sections of rewrites that are executed in sequential order after parsing. In the above example, the two rewrites are responsible for replacing a `fact` term with its actual value by unfolding factorials into multiplications. At the time of applying these rewrites, the MetaPRL refiner contains several "built-in" conversions that, for example, reduce the meta arithmetic terms.
- *Optimizations*: Optional target patterns for optimizations.

Given our language module, `1+2+3+4!` yields `Itt_int_base!number[30:n]`, a MetaPRL number term representing the number 30.

4 System Architecture

There are several technical issues in integrating these systems. The first issue is defining a shared language for MetaPRL and Mojave (Phobos and MetaPRL already share a common language of terms). Next, in order for MetaPRL to reason about Mojave programs, we have to formalize the language, including its operational semantics.

4.1 FIR as a common language

We are using the FIR as the common MetaPRL/Mojave language, for several reasons. First, all the front-ends, including C, ML, and Java produce programs in FIR; if we can reason about the FIR, we can reason about programs produced by any of these languages. Second, the FIR has a precise semantics, where many of the source languages do not (for example, C).

However, the disadvantage of using the FIR is that it is difficult in general to translate source-level specifications and proofs to their corresponding specifications and proofs at the FIR level. The optimization problem is not nearly so hard, and much of our current work has been developing operational reasoning in MetaPRL about programs in the FIR.

The Mojave compiler does not use terms for its internal representation of programs. For communication with MetaPRL we develop "glue" code to translate between the Mojave FIR representation of a program, and the MetaPRL term representation of the program. This glue code is straightforward; for the remainder of the paper, we will assume programs are represented as terms.

4.2 FIR term representation

The MetaPRL term representation for FIR programs is straightforward. In most cases, the term that represents an FIR expression has an explicit operator name (opname), and a set of subterms described recursively. We illustrate the translation with a few examples, using the notation $[\![\cdot]\!]$ for the term representation of a FIR program.

The atom values tagged with a name and any additional parameters.

$$\begin{aligned}
[\![v]\!] &= \mathsf{atomVar}\{v\}\\
[\![i]\!] &= \mathsf{atomInt}\{i\}\\
[\![i_{32,signed}]\!] &= \mathsf{atomRawInt}\{\mathsf{int32};\ \mathsf{signedInt};\ i\}
\end{aligned}$$

Expressions are a bit more interesting because of their binding structure. The term representation of an expression, in contrast to the ML representation, uses explicit binding in the form of higher-order abstract syntax [15]. As Pfenning mentions, the advantage of higher-order abstract syntax is that substitution and alpha-renaming are automatic. The disadvantage is that analyses that modify the binding in unusual ways become difficult to define. We illustrate the term syntax with the term for binary arithmetic.

$$[\![\mathbf{let}\, v\colon t = a_1\ binop\ a_2\ \mathbf{in}\ e]\!] = \mathsf{letBinop}\{[\![t]\!];[\![binop]\!];[\![a_1]\!];[\![a_2]\!];v.[\![e]\!]\}$$

The remaining terms follow the same general form, and we omit them here.

4.3 Operational semantics

The operational semantics of the FIR is defined using rewriting rules in MetaPRL. The actual operational definition is quite large because there are many combinations of arithmetic operations and values. However, the forms of definition are straightforward. For example, the operational rule for addition has the following general form, which we write using simplified pretty-printed notation. To be faithful to the implementation, we are using modular arithmetic.

$$(\mathbf{let}\, v\colon t = \mathsf{atomInt}\{i\} + \mathsf{atomInt}\{j\}\ \mathbf{in}\ e[v]) \rightarrow e[i+j]$$

The control operator match has a more interesting definition. The match operator is a pattern match of a number i against a set of intervals. The number of interval cases is arbitrary, and reduction performs one case analysis at a time.

$$\begin{aligned}
&(\mathbf{match}\, i\, \mathbf{with}\, s \rightarrow e \mid cases)\\
\longrightarrow\ &(\mathbf{if}\ i \in s\ \mathbf{then}\ e\ \mathbf{else}\ \mathbf{match}\, i\, \mathbf{with}\ cases)
\end{aligned}$$

The interval s is represented as a list of closed intervals $[i_{11}, i_{12}], \ldots, [i_{n1}, i_{n2}]$, and the membership operation is defined inductively.

$$\begin{aligned}
(i \in ([j,k] :: interval)) &\rightarrow (i \leq j \wedge i \leq k) \vee (i \in interval)\\
(i \in []) &\rightarrow false
\end{aligned}$$

Once again, the remainder of the operational semantics is straightforward, and we do not present it here.

4.4 Models and usage

The question of *models* is probably the most interesting topic in this translation. Ideally, we would develop a model of the FIR in a type theory or other higher-order logic, and then *prove* the operational semantics and typing rules. Note that a complete model would need to represent both partial functions and general recursive types. We have not developed this model, and we presume that it is likely that we will need to restrict validation to a fragment of the calculus that has a well-defined formal model. In the meantime, we treat the operational semantics axiomatically.

As Figure 1 illustrates, there are currently two major ways that we use the MetaPRL/Mojave system. The (i) path uses the Mojave front-ends to generate FIR code, which is than passed to MetaPRL for optimization. The (ii) path produces FIR from a Phobos description, optimizes it, and passes it to the compiler for code generation.

5 Examples

We illustrate the system with two optimizations. The MetaPRL/Mojave systems, including examples, can be downloaded from `http://www.metaprl.org` and `http://mojave.cs.caltech.edu`.

5.1 Dead-code elimination

Some standard code transformations are incredibly easy to define using term rewriting. Dead-code elimination is one of the simplest. The idea of dead-code elimination is to remove any code that does not affect the result of a computation. The problem is not computable in general, although we can develop proof procedures to catch a fairly broad set. The usual approximation is to use a *syntactic* characterization: the sub-expression e_1 in $\mathbf{let}\, v\colon t = e_1 \,\mathbf{in}\, e_2$ is *dead* if v is not free in e_2. Second-order term rewriting makes this easy to characterize. The following rewrites can be derived as theorems in MetaPRL:

$$\begin{aligned}
(\mathbf{let}\, v\colon t = \mathit{unop}\, a \,\mathbf{in}\, e) &\to e \\
(\mathbf{let}\, v\colon t = a_1\ \mathit{binop}\ a_2 \,\mathbf{in}\, e) &\to e \\
(\mathbf{let}\, v = \mathit{alloc} \,\mathbf{in}\, e) &\to e \\
(\mathbf{let}\, v\colon t = a_1[a_2] \,\mathbf{in}\, e) &\to e
\end{aligned}$$

Dead-code elimination is then performed by normalizing the program with respect to these rewrites. Note that the expression e in the redex does not mention the variable v, which means that v is not allowed to appear free in e (the second-order pattern $e[v]$ would have allowed v to appear in e, and would not be provable [13]). Also, note that the first-order definition using substitution would not be as useful for dead-code elimination because the rule does not specify explicitly that the variable v is dead.

$$(\mathbf{let}\, v\colon t = \mathit{unop}\, a \,\mathbf{in}\, e) \to e[a/v]$$

5.2 Partial evaluation

The next example illustrates a simple, but non-trivial, application of partial evaluation. Consider the following FIR code (we omit the types for clarity). The *power* function computes the value $res * x^y$, and passes it to the continuation *cont*. The *power5* function computes the specific case where $res = 1$ and $y = 5$.

```
let power (res, x, y, cont) =
    if y = 0 then
        cont(res)
    else
        let res' = res * x in
        let y' = y - 1 in
            power(res', x, y', cont)

let power5 (x, cont) =
    power(1, x, 5, cont)

inline power(res, x, number[i], cont)
```

For this example, we would like to "unroll" the definition of *power5* to a sequence of 4 multiplications $x*x*x*x*x$. The programming language, defined in Phobos, includes the **inline** extension where the programmer can indicate *patterns* that should be expanded using the **inline** keyword. For the example above, the inline instruction specifies that a call to the *power* function should be expanded when its third argument is a number.

Based on this information, the MetaPRL system constructs a rewrite to force the unfolding.

```
let power (res, x, number[i], cont) ⟶
    if number[i] = 0 then
        cont(res)
    else
        let res' = res * x in
        let y' = number[i] - 1 in
            power(res', x, y', cont)
```

Next, *power5* is normalized relative to the rewrite, and all calls to the *power* function with a constant exponent are inlined. The final definition of the *power5* function is as follows.

```
let power5(x, cont) =
    let x1 = x * x in
    let x2 = x1 * x in
    let x3 = x2 * x in
    let x4 = x3 * x in
        cont(x4)
```

The optimized code produced by MetaPRL is still suboptimal; if we assume that multiplication is associative, the number of multiplications can be reduced to three. This example, however, illustrates the general principle.

6 Conclusion and Future Work

The work presented in this paper is demonstrates the *principle* of formal integrated design environments, but the integration is far from complete. Among the next steps are: 1) a complete formalization of the operational semantics and type system for the FIR, 2) a more general framework for performing partial evaluation. The Mojave compiler architecture has many (around 30) stages of program transformation and optimization. It seems likely that many of these transformations can be significantly simplified by implementing them in the theorem prover. Another important direction is “bootstrapping.” Currently, MetaPRL is still layered over the OCaml compiler because the Mojave implementation of ML does not include a module system. Removal of this obstacle would enable complete integration of the formal design environment.

References

1. Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 215–224. IEEE, June 1987.
2. Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proc. of Fifth Symp. on Logic in Comp. Sci.*, pages 95–197. IEEE, June 1990.
3. Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1998. Second Edition.
4. Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, April 1995. `http://www.csl.sri.com/sri-csl-fm.html`.
5. M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.
6. Jason Hickey. Fault-tolerant distributed theorem proving. In Harald Ganzinger, editor, *Automated deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 227–231, Trento, Italy, July 7–10, 1999. Springer-Verlag.
7. Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, 2001.
8. Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`, 2002.
9. S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994.
10. S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

11. Christoph Kreitz, Mark Hayden, and Jason Hickey. A proof environment for the development of group communication systems. In C. Kirchner and H. Kirchner, editors, *15^{th} International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 317–332. Springer Verlag, 1998.
12. Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17^{th} ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 34 of *Operating Systems Review*, pages 80–92, 1999.
13. Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In *Theorem Proving in Higher-Order Logics (TPHOLs '02)*, 2002.
14. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer LNCS 828, 1994.
15. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.
16. Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.