推薦論文● PPL2007

IIR: an Identifier-based Intermediate Representation To Support Declarative Compiler Specifications

Eiichiro Chishiro

For compiler developers, one big issue is how to describe a specification of its intermediate representation (IR), which consists of various entities like symbol tables, syntax trees, analysis information and so on. As IR is a central data structure of a compiler, its precise specification is always strongly desired. However, the formalization of an actual IR is not an easy task since it tends to be large, has complex interdependency between its entities, and depends on a specific implementation language. In this paper, as a first step to solve this problem, we propose a new data model for IR, called IIR. The goal of IIR is to describe a specification of IR declaratively without depending on its concrete implementation detail. The main idea is to model all entities of IR as relations with explicit identifiers. By this, we can develop an IR model transliterally from an actual IR, and describe its specification by using the full expressiveness of conventional logic languages. The specification is inherently executable and can be used to check the validity of IR in compile time. As a practical case study, we formalized an IR of our production compiler in IIR, and developed a type system for it in Prolog. Experimental results about size and performance are shown.

1 Introduction

Developing a production compiler is a hard work. The target architecture continues to evolve. New language features continues to be added. To keep up with these changes and generate efficient compiled codes, we should continue to implement new features for analyses, optimizations, code generations and more. As a result, an actual implementation of a production compiler tends to be quite large — sometimes over a million of lines.

A compiler consists of a series of modules to realize its features. Most modules work on the intermediate representation (IR), which is a source program representation in the compiler. For instance, a module for dataflow analysis analyzes the IR and adds dependence information to it. A module for dead code elimination traverses the IR and removes unnecessary statements from it. Thus the IR works as a common interface between compiler modules, and its specification judges the correctness of each module.

Unfortunately, the complete specification of the IR rarely exists in practice. The IR continually changes in the development process, and it is often the case that given some IR, developers disagree with its precise meaning, or worse, whether it is legal (well-formed).

This severely undermines the quality of a compiler. You are forced to develop your module based on an obscure specification of the IR. For instance, if you want to know all the patterns coming into your module, you need to dive into massive source codes and examine all possibilities. This makes the development process slower and less reliable.

In research on the semantics of programming languages, there exist lots of systematic ways to define the specification of a programming language [41]. Generally, we first define the syntax of program-

宣言的なコンパイラ仕様記述を支援するための識別子にもとづく中間表現.

千代英一郎, 日立製作所システム開発研究所, Hitachi Systems Development Laboratory.

本論文は, 第 9 回プログラミングおよびプログラミング言語ワークショップ (PPL2007) の発表論文をもとに発展させたものである.

コンピュータソフトウェア, Vol.25, No.3 (2008), pp.113–134.

[研究論文] 2007 年 4 月 30 日受付.

ming languages in BNF, and then static/dynamic semantics based on the syntactic structure of programs. If we could regard the IR as a programming language, we would follow the same approach.

This approach fits very well with methods based on declarative programming languages (here we call these 'declarative methods'). The specification written in a declarative language can serve as both human-readable document and machine-executable checker. When writing the specification of the IR, the latter property is very useful for relating the specification to its implementation.

Besides specifications, there exist many declarative methods for compiler features such as program analyses [11] [18] [38], optimizations [4] [12] [26] [39] and code generations [19]. These declarative methods are promising approaches to reducing the burdens of compiler development.

Problem However, these methods cannot be applied to the existing IR in a straightforward way. The main difficulty lies in the difference between the IR of an actual compiler and the program representations used in these methods.

The IR has several aspects which are difficult to be treated in declarative methods. First, the IR forms a graph-structured data where entities refer to each other (e.g., control flow graphs), or share a common entity (e.g., symbol tables). Next, IR entities of a production compiler tend to have a large number (sometimes more than a hundred) of attributes to carry miscellaneous information such as source line numbers, analysis results, compiler directives and architecture-specific extensions. Finally, the situation becomes more complicated when the implementation types of IR entities are optimized intricately to get better performance of compilation.

In contrast, in most declarative methods, programs are modeled as tree-structured terms suitable for treatment in declarative languages. Many attributes of programs are abstracted to focus on the essence. Thus, to apply such method to the IR, we first need to extract a model of the IR by translation. As program models usually differ between methods, we have to do this for each method. Developing such a transformation is tedious and error-prone work, and sometimes impossible due to the substantial difference between the IR and the assumed program representation of the method.

Goal Our goal is to provide a way to fill these gaps between the existing IR in practice and declarative methods in research. What we want is an IR model which

1. can faithfully represent the structures of an actual IR, and
2. can also be a basis on which declarative methods are developed directly.

Such models enable us to write the specification of IR in declarative languages, which has many benefits. The most notable one is executability as said above, which is invaluable and indispensable when developing a large formal specification. We can be confident in the correctness of a specification by executing it on IR models of test programs. If a model is a faithful representation of IR, the distance between an actual implementation and its model is sufficiently small, and the results obtained from a model are sure to also hold on an implementation. In addition, existing declarative methods can be built directly on an IR model, which releases us from the tiresome work of translation for each method.

In a sense, our motivation is similar to that of the researchers and developers in the area of databases. They also aim to create a universal data model upon which any application software can be developed. One of the most successful achievements has been the relational data model (RDM) [9]. While using RDM as an IR model seems an attractive approach, it has some shortcomings in our situation, particularly in the interaction with declarative languages. We will discuss this issue in Section 5.

Proposal To achieve our goal, we have developed a new IR model, which we call an *identifier-based intermediate representation* (*IIR*). Our key insight in developing an IR model is that major difficulties described above stem from the anonymity of IR entities. While each entity has its identity in an actual implementation, there is no counterpart in conventional program models used in declarative methods. This discrepancy is an impediment to faithfully modeling an actual IR structure.

The essence of IIR is as follows:

1. all entities have an explicit identifier.
2. all entities are represented as relations.

The first condition means that we give each entity a unique identifier and always use this when referring

to the entity †1.

Note that the meaning of the term *identifier* here is not limited to names of variables or functions in programming languages. For instance, in an expression `x + x`, each operand of `+` has same variable identifier `x`, but they are not the same entity instance and each of them has its own identity. The identifiers in IIR provide an explicit and abstract way of representing the identity of each entity. This solves the sharing problem discussed above as we can always distinguished the entity itself form references to it by identifiers. An example for this is shown in Section 2.

The second condition means that all entities in IIR are relations in the mathematical sense. As relations can be easily treated as predicates in logic, we can write a strict well-formedness condition for the IR in declarative languages to prevent excessive flexibility caused by the identifiers. This solves the model-extraction problem discussed above as we can now apply declarative methods directly on the IR itself, not an extracted model of the IR. There is no need for translation, and all information of the IR is exposed to those methods. The formal definition of IIR is given in Section 2.

Evaluations To evaluate the effectiveness of our IIR model, we performed case studies using our production compiler. It is a C/C++ compiler for our embedded RISC processors based on our multi-language/multi-platform common compiler framework which supports several source languages such as C/C++ and Fortran. It has been continually developed over ten years, and the lines of code written in C now number over 1.5 million. We believe that these characteristics make it a suitable candidate for evaluating our method.

As a case study, we developed an IIR model of a high-level IR of our compiler called MIR, and a formal specification of MIR as a type system based on the model (Section 3). We performed two experiments on the specification. First, we ran the specification on hundreds of test programs to confirm that the specification did not reject the correct IR, i.e., that the specification was experimentally complete. The statistical results of the specification clearly showed that the scale of the full specification is quite large, and without such testing it would be virtually impossible to write a correct specification conforming to an actual implementation.

Next, we used the specification as an IR checker (lint). We ran it on some IRs that were incorrect because of bugs in an older version. We found that two bugs reported on our web site [28] [29] could be detected as type errors, although the causes of these bugs had nothing to do with types. While the effectiveness of an IR checker is well known in research [21], our result can be used as a practical example to support this claim in industry.

As another case study, we developed a reaching definition analysis in a declarative language on an IIR model of MIR (Section 4). The analysis was obtained through a straightforward translation of a classical dataflow equations [2], but worked on an actual IR model. While the issue of aliasing is often ignored in papers to simplify the presentation, the ability to handle aliasing is indispensable. We show that aliasing can be incorporated straightforwardly in an IIR model.

Contributions Our contributions can be summarized as follows:

- We developed a new IR data model, called IIR, which allows us to faithfully model IR in an actual implementation and describe a precise specification of it in conventional declarative languages. IIR also makes it possible to straightforwardly apply many existing declarative methods of program analysis, optimization, and code generation to a full IR, not a simplified subset of it.
- We confirmed the practicality of our method by applying it to our large-scale production compiler. We developed an IIR model of a high-level IR of our compiler and its specification as a type system in Prolog. We tested the specification on hundreds of test programs and confirmed its correctness. We also found that two publicly reported bugs could be detected by using the specification as a validity checker.

In another respect, our work in total can be seen as a case study of applying well known declarative methods to a real, large software. Our results show that existing declarative methods can be suf-

†1 The term *entity* is ambiguous. To be precise, we should distinguish an entity and an entity set, which consists of all "similar" entities. But as it is clear from the context in most cases, we usually use the term *entity* for both meanings.

ficiently practical in industry with a little bit of insight about the way to model software.

2 IIR

IIR is a data model that is designed for a compiler IR. It provides a means to describe precisely the set of data which we would regard as valid IR.

The way of defining IIR is very close to that of the relational data model (RDM) [9]. In RDM, we use a relation schema to model data. Each relation schema defines the set of its instances, which are usually called relations. Similarly, in IIR, we use an IIR schema to model an IR. Each IIR schema defines the set of its instances, which we call IIR schema instances. For those unfamiliar with RDM, Appendix B contains a brief summary of RDM.

The set of valid IR is usually a subset of the set of IIR schema instances. IIR has a strong way of defining this subset, that is described in Section 2. 3.

2. 1 Definitions

We begin with basic definitions of IIR. The intuition of the definitions will be explained later with examples. A discussion of design decisions will be deferred in Section 5.

Here are the definitions of an IIR schema and its instances.

Definition 1 (IIR schema). Let I be a set of identifiers, C be a set of class names, A be a set of attribute names, and V be a set of values. IIR schema R is a tuple (I, C, A, V, $class$, $attr$, dom) such that $class : I \rightarrow C$, $attr : C \rightarrow \wp(A)$, $dom : A \rightarrow \wp(V)$, $I \cap C = C \cap A = \phi$ and $I \subseteq V$.

We refer to each component of R as I^R, C^R, A^R, V^R, $class^R$, $attr^R$ and dom^R. For convenience, we usually omit a component $class$ in an IIR schema definition. We also omit some components such as A and V in an IIR schema if these omitted components are clearly implied from the context. □

Definition 2 (IIR schema instance). Let R be an IIR schema (I, C, A, V, $class$, $attr$, dom). IIR schema instance r of R is a set of tuples such that $r = \cup_{i \in I'} r(i)$, where $I' \subseteq I$ and $r(i)$ satisfies the following:

1. $r(i) \subseteq$
   $\{(i, a, v) \mid a \in attr(class(i)), v \in dom(a)\}$.
2. $\forall i, a, v_1, v_2.$
   $(i, a, v_1) \in r(i) \wedge (i, a, v_2) \in r(i) \Rightarrow v_1 = v_2$.

We call an IIR schema instance an IIR instance where this is not ambiguous. □

Condition 2 means that each v in (i, a, v) is functionally dependent on i and a. Note that in definition 2, we do not require that an identifier i of class c must have all attributes of c, i.e., all attributes are optional. We will show later how we can specify a valid set of attributes for each i.

Example 1 (ExpL). Consider a simple expression language $ExpL$, whose abstract syntax is given below.

$$\begin{array}{lll} n & \in & \mathit{IntegerLiterals} \\ x & \in & \mathit{Variables} \\ e & ::= & n \mid x \mid e{+}e \end{array}$$

A straightforward definition of an IIR schema E for $ExpL$ is as follows (A, V and $class$ are omitted as mentioned above.):

$$\begin{array}{lcl} I^E & = & \mathit{Integer} \\ C^E & = & \{\texttt{con}, \texttt{var}, \texttt{add}\} \\ attr^E(\texttt{con}) & = & \{\texttt{kind}, \texttt{value}\} \\ attr^E(\texttt{var}) & = & \{\texttt{kind}, \texttt{name}\} \\ attr^E(\texttt{add}) & = & \{\texttt{kind}, \texttt{child1}, \texttt{child2}\} \\ dom^E(\texttt{kind}) & = & C^E \\ dom^E(\texttt{name}) & = & \mathit{String} \\ dom^E(\texttt{value}) & = & \mathit{Integer} \\ dom^E(\texttt{child1}) & = & dom^E(\texttt{child2}) = I^E \end{array}$$

For example, consider an expression $(\texttt{x} + \texttt{x}) + \texttt{100}$. This is represented as an IIR instance r_e of $ExpL$.

$$\begin{array}{ll} r_e = & \{(1, \texttt{kind}, \texttt{add}), (1, \texttt{child1}, 2), (1, \texttt{child2}, 3), \\ & (2, \texttt{kind}, \texttt{add}), (2, \texttt{child1}, 4), (2, \texttt{child2}, 5), \\ & (3, \texttt{kind}, \texttt{con}), (3, \texttt{value}, 100), \\ & (4, \texttt{kind}, \texttt{var}), (4, \texttt{name}, \text{“}\texttt{x}\text{”}), \\ & (5, \texttt{kind}, \texttt{var}), (5, \texttt{name}, \text{“}\texttt{x}\text{”}) \quad \} \end{array}$$

Entity 1 represents the root of the expression (the second +). Entity 2 and 3 represent the root of its subexpressions (the first + and 100). Entity 4 and 5 represent leaf expressions (x) of x + x (entity 2). □

In the above example, each occurrence of x has all the attributes of the variables: that is, all attributes

of x are repeated twice (In this case, a string literal "x" is duplicated). In general, a variable has lots of attributes, and it is well known this kind of redundancy is likely to violate data consistency.

In the usual IR, information regarding the same variable is shared by all of its occurrences in the form of 'symbol tables'. A variable occurring in expressions does not hold information itself, but *refers to* its symbol table entry. Constant literals which denote the same value are also shared.

In IIR, thanks to the property that all entities have explicit identifiers, such sharing structure can be represented straightforwardly.

Example 2 (ExpL with sharing). We first introduce an IIR schema S for symbol tables as follows:

$$\begin{array}{lcl} I^S & = & \mathit{Integer} \\ C^S & = & \{\texttt{con}, \texttt{var}\} \\ \mathit{attr}^S(\texttt{con}) & = & \{\texttt{kind}, \texttt{value}\} \\ \mathit{attr}^S(\texttt{var}) & = & \{\texttt{kind}, \texttt{name}\} \\ \mathit{dom}^S(\texttt{kind}) & = & C^S \\ \mathit{dom}^S(\texttt{name}) & = & \mathit{String} \\ \mathit{dom}^S(\texttt{value}) & = & \mathit{Integer} \end{array}$$

Accordingly, we change an IIR schema E to refer to S:

$$\begin{array}{lcl} I^E & = & \mathit{Integer} \\ C^E & = & \{\texttt{con}, \texttt{var}, \texttt{add}\} \\ \mathit{attr}^E(\texttt{con}) & = & \{\texttt{kind}, \texttt{sym}\} \\ \mathit{attr}^E(\texttt{var}) & = & \{\texttt{kind}, \texttt{sym}\} \\ \mathit{attr}^E(\texttt{add}) & = & \{\texttt{kind}, \texttt{child1}, \texttt{child2}\} \\ \mathit{dom}^E(\texttt{kind}) & = & C^E \\ \mathit{dom}^E(\texttt{sym}) & = & I^S \end{array}$$

The expression $(\texttt{x} + \texttt{x}) + \texttt{100}$ of example 1 is now represented as IIR instances r_e of E and r_s of S.

$$\begin{array}{ll} r_e = \{ & (1, \texttt{kind}, \texttt{add}), (1, \texttt{child1}, 2), (1, \texttt{child2}, 3), \\ & (2, \texttt{kind}, \texttt{add}), (2, \texttt{child1}, 4), (2, \texttt{child2}, 5), \\ & (3, \texttt{kind}, \texttt{con}), (3, \texttt{sym}, 1), \\ & (4, \texttt{kind}, \texttt{var}), (4, \texttt{sym}, 2), \\ & (5, \texttt{kind}, \texttt{var}), (5, \texttt{sym}, 2) \quad \} \\ r_s = \{ & (1, \texttt{kind}, \texttt{con}), (1, \texttt{value}, 100), \\ & (2, \texttt{kind}, \texttt{var}), (2, \texttt{name}, \text{“x”}) \quad \} \end{array}$$

□

Adding extra information, e.g., analysis results, to *ExpL* takes little effort. For example, if we want to express dependence information between variables (here we assume that an assignment statement is also added to *ExpL*), we just define an IIR schema F of attributes defpoint and usepoint whose domains are I^E.

$$\begin{array}{lcl} I^F & = & \mathit{Integer} \\ C^F & = & \{\texttt{flowdep}\} \\ \mathit{attr}^F(\texttt{flowdep}) & = & \{\texttt{defpoint}, \texttt{usepoint}\} \\ \mathit{dom}^F(\texttt{defpoint}) & = & I^E \\ \mathit{dom}^F(\texttt{usepoint}) & = & I^E \end{array}$$

By analogy with the relational data model, we can define IIR database.

Definition 3 (IIR database). Let R_1, R_2, ..., R_n be IIR schemas. IIR database D of these is a tuple of IIR schemas $\Pi_i R_i = (R_1, \ldots, R_n)$. □

Definition 4 (IIR database instance). Let D be an IIR database $D = \Pi_i R_i$. IIR database instance of d is a tuple $\Pi_i r_i$ such that each r_i is an instance of R_i. □

An IR can be modeled as an IIR database D. For Example 2, we can model the IR as database schema $D = (E, S)$ and its instance as a database instance (r_e, r_s) of D.

2.2 IIR Representation of Various Data Structure

In this section, we show how major data structures of IR can be represented in IIR.

2.2.1 Record

A record is one of the most basic data structures of IR, and is implemented by a structure type in C, a class in C++ or Java, or a record type in ML.

Since a record is essentially a labelled product type, we can straightforwardly represent a record structure as a class in IIR. Let τ be a record with fields $\prod_{i=1}^{n} l_i : \tau_i$ where each l_i is a field of type τ_i. The corresponding IIR class c can be defined as follows:

$$\begin{array}{lcl} \mathit{attr}(c) & = & \{l_1, ..., l_n\} \\ \mathit{dom}(l_i) & = & \mathit{dom}(\tau_i) \quad \text{for all } i \end{array}$$

where $\mathit{dom}(\tau_i)$ on the right-hand side is a domain corresponding to τ_i.

2.2.2 Variant

A variant type is also a basic data structure of IR. This is a labelled sum type, and in IR, it is used to represent the *is-a* relation between entities. For example, a syntactic category "Exp" of an intermediate code usually has several kinds of entity like "add", "sub", and so on. A variant type can be

implemented as a union type in C, subclasses of a class "Exp" in C++ or Java, or an algebraic data type in ML.

In IIR, we can represent a variant type by adding an arbitrary but distinct attribute name to represent the label of each entity. Let τ be a sum $\sum_{i=1}^{n} l_i : \tau_i$ where each l_i is a content type τ_i. The corresponding IIR class c can be defined as follows:

$$\begin{array}{lcl} attr(c) & = & \{\texttt{kind}\} \cup \bigcup_i attr(c_i) \\ dom(\texttt{kind}) & = & \{l_1, ..., l_n\} \end{array}$$

where `kind` is an attribute name not occurring in $attr(c_i)$ †2 and c_i is a corresponding IIR class of τ_i. The attribute set of each entity i of class c in IIR instance r depends on the value v of `kind`: i.e., $(i, \texttt{kind}, v) \in r$.

Note that in contrast with the algebraic data type in ML, the variant type of IIR enables *flat embedding*. The subclass relation can be treated similarly.

2.2.3 Array

A single-dimensional array (e.g., of index range $0, \ldots, n$) can be represented as a record type with numeric labels $0, \ldots n$, corresponding to each index values.

For a multi-dimensional array type (e.g., m-dimensional), there are two ways to represent it. One is to regard it as a single-dimensional array of $(m-1)$-dimensional array type. By doing this process recursively, we just need to treat a single-dimensional array. The other is to introduce attributes $\texttt{dim}_\texttt{i}$ for each dimensions i $(1 \leq i \leq m)$, and identify each elements of an array by a combination of index values of all dimensions.

For the latter, the corresponding IIR class c can be defined as follows:

$$\begin{array}{lcl} attr(c) & = & \{\texttt{dim}_i | 1 \leq i \leq m\} \cup attr(c_e) \\ dom(\texttt{dim}_i) & = & \{l_i, ..., u_i\} \end{array}$$

where $\texttt{dim}_i$ is an attribute name not occurring in $attr(c_e)$, c_e is a corresponding IIR class of τ_e, and l_i (u_i) is an upper (lower) bound index of the ith-dimension.

For example, an array type `int[2][3]` in C is represented as follows:

$$\begin{array}{lcl} attr(c) & = & \{\texttt{dim}_1, \texttt{dim}_2\} \cup attr(\texttt{int}) \\ dom(\texttt{dim}_1) & = & \{0, 1\} \\ dom(\texttt{dim}_2) & = & \{0, 1, 2\} \end{array}$$

2.2.4 List

A linked list is a basic data structure which has *references* between entities. In most languages, references are explicitly or implicitly implemented by memory addresses. In IIR, we can realize this straightforwardly by using identifiers as symbolic addresses. This is the key property of IIR, providing a foundation for most of our method.

For example, the integer list of class c can be realized as follows:

$$\begin{array}{lcl} attr(c) & = & \{\texttt{value}, \texttt{next}\} \\ dom(\texttt{value}) & = & \mathit{Integer} \\ dom(\texttt{next}) & = & I \end{array}$$

where I is a set of identifiers. Each element except the last one of a list has a tuple of the form $(i, \texttt{next}, i')$ where i' is the identifier of the next element. The last element of a list does not have a tuple $(\_, \texttt{next}, \_)$, which implies the absence of its successor.

Example 3 (Tree). A tree data structure can be seen as a variant type which has references of one to many. If the arities of all nodes are fixed, we can represent a tree by using attributes such as $\texttt{child}_1, ..., \texttt{child}_n$ where n is the max arity of all nodes. If this is not the case, we can represent a tree as follows:

$$\begin{array}{lcl} attr(c) & = & \{\texttt{parent}, \texttt{ith}\} \\ dom(\texttt{ith}) & = & \mathit{Integer} \\ dom(\texttt{parent}) & = & I \end{array}$$

where `ith` represents that the entity is a i-th child of a parent. This is a common way of treating 1-n relations in RDM. □

Example 4 (Graph). We can represent a graph data structure (n-n relation) in almost the same way as RDM. That is, we introduce a new class which represents each relationship between graph nodes:

$$\begin{array}{lcl} attr(c) & = & \{\texttt{from}, \texttt{to}\} \\ dom(\texttt{from}) & = & dom(\texttt{to}) = I \end{array}$$

where c' is a class of graph nodes, and c is a class of graph edges. We can add various kinds of attribute

†2 In case study (Section 3), we maintained a name table by hand for all IIR schema to avoid attribute name-clash. It is desirable to develop a formal language of specifying IIR schemas, and check them automatically.

to edges such as execution probabilities. □

2.3 Validity of IIR

As mentioned in Section 1, precise specification of an IR helps us develop a compiler in an efficient and reliable way. When we develop a module of a compiler, it is necessary and sufficient to take care of all possible input IR patterns to the module which are valid with respect to the specification. Thus, it is desirable that the IR specification contains all patterns which we want to regard as valid and also *does not contain* other patterns which we do not want to regard as valid.

While a pervasive approach is to describe the IR validity conditions based on types or some kinds of data schema, this is often insufficient and gives only a coarse specification. For example, the syntax of expressions is often given as a BNF, which is a data schema for describing well-formed expressions. In most cases a BNF only gives coarse conditions for valid expressions, and we need more elaborate frameworks: e.g., type systems to describe the validity condition more precisely. To represent complex conditions, we often need to extend such frameworks by, for example, incorporating dependent types or refinement types into type systems. However, this is a highly theoretical task, requiring much effort and often impossible in practice.

In IIR, we describe complex validity conditions in conventional logic programming languages since all IIR instances are just tuples of values and can be regarded as predicates. We can define an IIR predicate of an IIR instance in the usual way.

Definition 5 (IIR predicate). Let r be an IIR instance of IIR schema R. IIR predicate r_p of r is a ternary predicate such that $r_p(i, a, v) \Leftrightarrow (i, a, v) \in r$. We usually omit the subscript of r_p and write r. These can be distinguished by the context. □

By using this predicative view, we can describe the validity conditions of IIR in a predicate logic. In this work we use a Prolog-style Horn clause form to specify conditions. Other logics are also possible, but this is sufficient for our cases.

As an example of this approach, we show the well-formedness condition of *ExpL* in Prolog, which is the most basic validity condition of IIR.

The well-formedness of the previous expression language *ExpL* can be defined as follows:

```
wf_exp(E) :-
  wf_exp_CON(E) ;
  wf_exp_VAR(E) ;
  wf_exp_ADD(E).
wf_exp_CON(E) :-
  exp(E,kind,con),
  exp(E,sym,S),
  wf_sym_CON(S).
wf_exp_VAR(E) :-
  exp(E,kind,var),
  exp(E,sym,S).
  wf_sym_VAR(S).
wf_exp_ADD(E) :-
  exp(E,kind,add),
  exp(E,child1,E1),
  exp(E,child2,E2),
  wf_exp(E1),
  wf_exp(E2),
  disjoint(E,E1),
  disjoint(E,E2),
  disjoint(E1,E2).

wf_sym_CON(S) :-
  sym(S,kind,con),
  sym(S,value,V),
  integer(V).
wf_sym_VAR(S) :-
  sym(S,kind,var),
  sym(S,name,V),
  string(V).
```

where `disjoint` is a predicate which holds if two operands do not share any entities and can be easily defined in Prolog. See appendix A for informal syntax and semantics of Prolog rules.

By this condition, IIR instances which contain the following tuples are regarded as ill-formed.

- A tuple which represents an add expression but is not linked to two valid children via `child1` and `child2`.
- A tuple which represents an add expression but its children shares some tuples representing their distinct but syntactically-equivalent subexpression.
- A tuple which represents a constant expression but is not linked to a constant symbol via `sym`.
- A tuple which represents a variable expression but is not linked to a variable symbol via `sym`.

As for typed IR, we can specify well-typedness conditions in a similar way. Suppose that all entities of *ExpL* have an additional attribute `type`, where $dom(\texttt{type}) = \{\texttt{int8}, \texttt{int16}, ...\}$. A well-typed condition of a constant symbol can be described as follows:

```
wt_sym_CON(E,T) :-
  sym(S, kind, con),
  sym(S,value,V),
  sym(S,type,T),
  valid_integer(V,T).
valid_integer(V,T) :-
  integer(V),
  (T == int8, -129 < V, V < 128; ...).
```

Note that a valid range of a constant value depends on its type. We can define such conditions easily without introducing elaborate type systems.

3 Case Study

In this section, as a practical case study of IIR, we show the result of applying our method to a large-scale production compiler.

It is a C/C++ compiler for our embedded RISC processors, written in C. Recent versions are based on our multi-language/multi-platform common compiler framework which supports several source languages such as C/C++ and Fortran, and target processors. Since this framework was first developed, it has been continuously evolved through the efforts of many people so that it can support new language features, analyses, optimizations, and target architectures. IR has also been extended to accommodate these functionalities.

Unfortunately, as it is often the case, this development process was done without defining a precise specification. As a result, the latest implementation instead forms a de facto specification, and developing any new functionality necessitates an in-depth scrutiny of massive source codes (over 1.5 million lines) to know implicitly-assumed rules for analyzing or transforming IR.

As a first step to remedy this, by using IIR, we have developed a type system for a high-level IR of our compiler, which provides more precise validity conditions of IR than implementation types or coarse BNF syntax rules. To do this, we first defined the IIR model of our IR, and base on it, developed a type system in Prolog.

In the reminder of this section, we show the detail of each steps.

Table 1 IIR schemas of MIR.

IIR schema	descriptions	$\|C\|$	$\|A\|$
CODE	code tree nodes	79	159
BB	basic blocks	11	26
EDGE	basic block edges	1	5
FUN	functions	2	119
SYM	symbols	9	123
CON	constant literals	4	41
ARR	arrays	1	10
DIM	array dimensions	1	11
NAME	names	1	5
SRC	source information	1	7

3.1 IIR Model

The first task is to define an IIR schema for each IR entity based on the existing implementation. In our compiler, a compilation mainly consists of two stages, one for source-level optimizations and the other for instruction-level optimizations. Each has its own IR: respectively, a high-level IR called MIR and a low-level IR called LIR. We chose MIR as the subject of this case study because it is more complex and difficult to formalize.

MIR consists of two parts: intermediate codes and several kinds of symbol tables. Table 1 shows the list of IIR schemas of MIR we developed. The $|C|$ column is the cardinality of a set C, i.e., the number of classes. Similarly, the $|A|$ column is the cardinality of a set A, i.e., the number of attributes.

Most of the schemas are self-explanatory. *NAME* is for the names of variables and functions. *SRC* is for debugging information such as source line numbers. *CODE*, *FUN*, and *SYM* have over one hundred attributes, that support our claim that the number of attributes tends to be large in a production compiler and thus must be administered in a systematic way.

Due to space limitations, here we discuss only one issue regarding this model extraction phase: the importance of modeling the implementation faithfully, not the essence of it.

Consider the type of nodes of intermediate codes in C. In a naive design, the node is implemented

as a structure type with a tag field and a union of types for specific node kinds:

```
struct N {
  Kind kind;
  union { Add add; Sub sub; ... } attr;
};
```

where a type `Kind` represents the kind of node, a type `Add` represents specific attributes for `add`, and the others are similar. This is the usual way of implementing variant types in C.

In a real implementation, though, several techniques are used for reasons of efficiency, and the design intention may be obscure. In MIR, since the number of kinds is less than 1024, a type `Kind` is implemented as a bitfield of 10 bits. This causes a padding of 22 bits if the next field is not a bitfield as in naive design. To avoid wasting memory, all node specific attributes like `Add` are divided into two groups, one that can be filled in bitfields of 22 bits and the others. This and other implementation issues complicate the model extraction and make it a non-trivial task.

We first tried to capture the intended essence of the design behind the complex implementation as described above and develop a clear model with informal transformation rules from the implementation to the model. However, this required us to perform the two different and complex tasks at the same time without any formal basis. As there are many attributes shown in Table 1, many design decisions must be made, which should not be done when exploring complex implementations as described above. Therefore, we finally decided to make the model reflect the structure of the existing implementation *as it was*, i.e. not considering semantic meanings.

IIR greatly helped us perform such a transliteration with little thought. We simply followed the encoding method described in Section 2 and developed an IIR model of MIR in a straightforward way. Each attribute could be treated individually and modeling could be done in parallel. Complex data like mutual referencing or recursive structures also cause no trouble in IIR, although such data necessitate extra effort in conventional algebraic data models.

We believe that the model extraction phase should be done as mechanically and uniformly as possible, which we can achieve in IIR for most cases.

3.2 CType

To specify the validity conditions of MIR, we take a conventional type-based approach. That is, we define the type system of MIR and represent validity conditions as well-typedness conditions. As MIR is a high-level IR of C, it is natural to define a type system of MIR based on a type system of C.

However, this is not so simple in practice since MIR is a common IR of multi-languages (C/C++ and Fortran) and its syntax and types do not completely match those of C. For example, an array type is represented in MIR as an untyped multi-dimensional array while an array type of C is a one-dimensional array of any data type, including an array type. All partially subscripted array expressions in C programs are represented by a fully subscripted one in MIR by supplementing `[0]` subscriptions.

Another example is the representation of member expressions. In MIR all contiguous member references for a nested structure are merged into one member (the innermost one). This is a naive form of scalar replacement [6] and enables us to similarly apply optimizations for scalar expressions to member expressions while it makes the MIR specification very complex, particularly by interacting with array subscriptions and indirections.

Since an IIR model of MIR faithfully reflects the actual structure of the existing implementation, as discussed in the previous sections, these gaps still exist between C and an IIR model of MIR.

To remedy this, we introduced one intermediate layer: that is, we developed an IIR model of types of C called CType. We also defined translation rules from the types of MIR to CType and a type system of MIR based on CType.

Tables 2 and 3 show the *attr* and *dom* of an IIR schema CT of CType. Compared to conventional models of the types of C found in the formal semantics of C, CT contains attributes of implementation-defined or nonstandard features such as the physical layouts of struct. Since the standard does not specify these, these are usually abstracted away and left unspecified. Of course, while this is rational for the purpose of defining the

Table 2 IIR schema CT of CType: *attr*.

class	attributes
var	kind, varOf
obj	kind, objectOf, qual
arr	kind, elementType, elementNum
int	kind, bitSize, sign
real	kind, bitSize, realKind
ptr	kind, bitSize, pointTo
rec	kind, bitSize, recordKind
member	kind, byteOffset, parentM, childM, memName
bitfield	kind, bitSize, bitOffset, declaredType
fun	kind, isVarArg
ret	kind, parentR, childR
arg	kind, parentA, childA, ith
noinfo	kind

Table 3 IIR schema CT of CType: *dom*.

attribute	domain
recordKind	{ struct, union }
realKind	{ float, double }
sign	{ signed, unsigned }
qual	{ volatile, const, restrict }
isVarArg	Bool
bitSize	Nat
bitOffset	Nat
byteSize	Nat
byteOffset	Nat
elementNum	Nat
ith	Nat
objectOf	I^{CT}
pointTo	I^{CT}
elementType	I^{CT}
varOf	I^{CT}
declaredType	I^{CT}
parent*	I^{CT}
child*	I^{CT}
kind	C^{CT}

static semantics of C, in the context of our work we need a more specific and concrete type system which can express all the MIR information. For instance, all integral types are defined by attributes of bitsize, signedness, and alignment information. The bitfield type has attributes of bit offset, bit size, byte offset, and declared type.

One benefit of IIR is that we can treat naturally incomplete types of C. As incomplete types lack some attributes, they can be treated as a supertype of the complete ones. In algebraic data types represented as a term tree, incomplete types are not allowed and must be defined as new types since the number of arguments for each constructor must match its arity. In contrast, as each entity is represented as a set of tuples, rather than as one data element, all incomplete types can be represented naturally as a subset of the complete types.

Another characteristic of CType is that it is designed to express implicit information in C as explicitly as possible. For instance, whereas *lvalue* and *rvalue* have semantically different meanings, this is not expressed in the types of C.

Let us consider the C code below.

```
int x;
x = x;
```

In C, both `x` in the assignment `x = x` have the same type `int`, though `x` in lhs is lvalue and `x` in rhs is rvalue.

This distinction is made explicit in CType. Let the identifier of CType of `x` in lhs be `i` and that in rhs be `j`. These CTypes `i` and `j` differ as follows (here we assume the bit size of `int` is 32):

```
{ ctype(i,kind,obj),
  ctype(i,objectof,j)
  ctype(j,kind,int)
  ctype(j,bitSize,32)
  ctype(j,sign,signed) }
```

where the `kind` of `i` is `obj` (lvalue) and that of `j` is `int` (rvalue).

This approach may remind you of the phrase type of Reynolds [30]. CType also resembles C types of Papaspyrou who also took a similar approach [25]. As this work has shown, this enables us to define typing rules in a clear and succinct way.

3.3 Type System of IIR Model

Given the above mentioned, we can define a type system of MIR in Prolog. Since MIR and CType are both IIR models, translation rules from MIR types to CType and typing rules can be defined in

Table 4 The size of a type system.

categories	#rules	#conditions
type environment generation (symbols)	26	218
type environment generation (constants)	7	39
typing rule for code tree nodes	86	543
ctype definitions	24	137
utilities	11	15

the same way as described in Section 2.

The whole system consists of two parts.

1. Construction rules of a type environment which assigns a CType to each symbol and constant.
2. Typing rules of assigning CTypes to MIR code tree nodes under the type environment constructed above.

One notable point of our approach, compared to the conventional term-based way of representing IR entities as compound terms, is the modification feasibility it enables. Suppose we add some attributes to an IR entity after our type system has been developed. In a term-based approach, this change may alter the arity of the term corresponding to the IR entity to which new attributes are added. This means that if we refer to subterms of the term in a type system, we should modify all these points to conform to the new arity. In contrast, with our IIR-based formalization, adding new attributes causes no modification to the existing system as it does not alter the arity of the term nor the type.

We can also utilize this feature to incrementally develop and refine a type system. That is, we can execute our system from the very initial phase of development, where only some of the attributes are treated. This is very convenient, particularly when the exact specification is not known *a priori*.

Ideally, this incremental refinement process must continue until the type system becomes fine enough to express all implicit assumptions we have on the IR. Of course, there is no way of checking this condition automatically, and we need to do coding reviews and/or tests (e.g., checking on the set of IR which we consider invalid) to be perfectly confident. Note that, for the purpose of using the type system as an IR checker, a partial specification is still useful to find some bugs, and as it becomes finer, we can find more bugs.

Table 4 shows the size of our type system. In a type system of MIR, there are 33 rules for part 1 (first and second rows), and 86 rules for part 2 (third and forth rows). In addition, there are 35 rules for both (fifth and sixth rows), in which 24 are for CType (equality, creation, etc).

For all the rules, the average number of predicates per a rule is 6.2. The largest, a rule for typing array symbols (in the first row of Table 4), consists of 29 predicates. This number informally suggests that an ad-hoc approach will be hard to scale to IR of a production compiler. When developing rules for each entity, we followed its IIR schemas and developed conditions in an attribute-wise way. The set of attributes in IIR schemas served as a check list, and prevented us from missing conditions for some attributes.

One interesting byproduct of this work is that we have found some entities in MIR which are syntactically different but have almost equivalent conditions of well-typedness. This suggests that there may be some redundancy in the MIR design. Conversely, we also have found that there are some entities which had several mostly-disjoint conditions of well-typedness. One example is an entity for indirect array expressions, which represents three patterns of C expressions, `p[i]`, `p->a[i]` and `p->q[i]` where `p`, `q` are pointers and `a` is an array. Though these have some commonality, the mostly-disjoint conditions of well-typedness suggest that it may be better to divide this kind of expression into three kinds of expressions.

3.4 Experimental Results

To confirm our specification does not reject correct IRs, we executed the specification of MIR on hundreds of test programs, including SPEC CInt95 benchmarks [31], EEMBC benchmarks [13] and Dhrystone 2.1.

Table 5 The size of IIR instances.

programs	lines	#tuples	*CODE*	*BB*	*EDGE*	*FUN*	*SYM*	*CON*	*ARR*	*DIM*	*NAME*	*SRC*
dhrystone	755	16,172	51.0	6.8	3.3	3.6	24.3	5.2	0.4	0.5	1.8	3.1
aifftr01	23,932	413,772	51.2	5.8	3.0	2.8	27.6	5.8	0.2	0.3	1.6	1.7
126.gcc	205,604	8,556,758	63.4	9.1	5.5	1.9	14.9	1.8	0.1	0.2	0.9	2.1

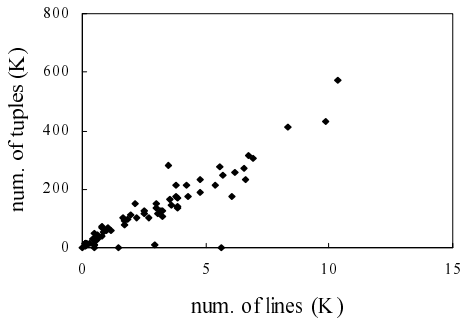

Fig. 1 The size of IIR instances in 126.gcc.

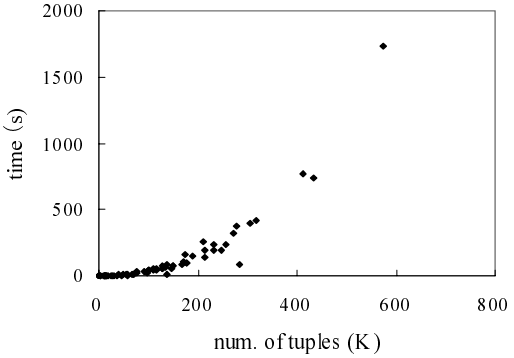

Fig. 2 Performance of type-checking.

Of course, such sampling tests cannot prove the completeness of the specification, but they provide some level of confidence. Actually, as many people have experienced, we have found many subtle specification bugs through the tests. And we now totally agree with the claim by Gordon et al. that "testing remains the only viable way of relating a specification to software of the complexity we were considering" [16].

Here, we only show some statistical results. To execute our specification on test programs, we implemented a type system of MIR in Prolog as described earlier. Each test program was compiled and its MIR was externalized into a file of IIR instances. Type checking was done by loading these two files in a SWI-Prolog 5.4.3 interpreter [34] and querying the predicate of well-typedness. The execution environment was Windows XP on a Pentium 4 (3 GHz) with 2GB memory.

Table 5 shows the size of the IIR instances with Dhrystone 2.1, aifftr01 from EEMBC and 126.gcc from SPEC CInt95 respectively representing small, medium, and large programs. The 'lines' column shows the number of lines of source files. The '#tuples' column shows the number of tuples of the IIR instances. Each number in the columns from '*CODE*' to '*SRC*' is a percentage representing the number of tuples of the corresponding IIR instance divided by the '#tuples' value. The distribution was clearly independent of the scale of the programs for the most part. The percentages of pure code part (*CODE*) are less than 70%, which indicates that we should not disregard non-code part in IR.

Figure 1 shows the size of the IIR instances in 126.gcc. Each point corresponds to a C source file in 126.gcc. The horizontal axis shows the number of lines of a source file; the vertical axis shows the number of all tuples of IIR instances of the file where K is 10^3. As we expected, the number of tuples was linearly proportional to the number of source file lines.

Figure 2 shows the elapsed time for type-checking 126.gcc. Each point corresponds to a C source file in 126.gcc. The horizontal axis shows the number of all tuples of IIR instances of a file; the vertical axis shows the elapsed time (seconds) of type-checking the file. The type-checking time is approximately quadratically related to the size of the IIR instances. We have not done any optimizations yet since performance is not of our primary concern.

The largest (573,736 tuples) and most time-consuming (1735.84s) file was `combine.c`, which was an instruction-level optimizer. As `global.c`, `stmt.c`, and `toplev.c` contained function calls in which argument types were not compatible with parameter types, we added a special rule for them in this evaluation. The effect of this modification

on performance is ignorable.

Soundness The above experiments concern the completeness of the specification. Regarding the soundness, we did an additional experiment. Besides simple testing using an artificially created ill-typed IIR, we checked two programs for which our previous compiler had generated incorrect executable code (the current compiler does not suffer from these problems, of course) [28] [29]. These were not bugs regarding types, but resulted from incorrect operations of the IRs; they emerged as ill-typed IRs, and our type system successfully identified the two bugs as type-errors. In functional languages community, it is well known that internal type-checking can detect many incorrect optimizations [35] [23]. Although our cases are trivial, this shows the importance of IR type-checking in a practical setting.

4 Other Applications

While our main purpose in developing a new IR model, IIR, has been to describe the precise specification of an IR, we can also use the realized model for other applications. Here we describe two of them.

4.1 Declarative Dataflow Analysis

Classical dataflow analysis can be straightforwardly described in declarative languages. Dataflow analysis generally consists of two phases:

1. Some dataflow facts are extracted from programs and reformulated for a specific analysis. For example, in a reaching definition analysis, all definition/use points are uniquely named by some naming schema and put into a predicate form. Name clashes are resolved by alpha-conversion.
2. These facts are iteratively propagated along control flows of a program until they reach the fixpoint.

Many researchers have proposed ways of specifying the second phase of analysis in declarative frameworks. In most cases they assume that dataflow facts and control flows are given in an appropriate form for their analyses, and concentrate on an efficient way of calculating the fixpoint for these.

For example, Ullman presents a simple reaching definition analysis in datalog [36] as follows:

```
clear(D,B) :-
  dvar(D,X), not(kill(X,B)).
out(D,B) :-
  gen(D,B).
out(D,B) :-
  in(D,B), clear(D,B).
in(D,B) :-
  out(D,C), succ(C,B).
reach(D,U) :-
  in(D,B), exposed(U,B),
  dvar(D,X), uvar(U,X).
```

where `dvar`, `uvar`, `kill`, `gen`, `exposed`, and `succ` are given as a result of the second phase of extracting dataflow facts.

While the first phase is regarded as a trivial task and rarely considered, this phase often needs a large amount of work when implementing an analysis, particularly in a production compiler. In fact, almost all of the bugs of dataflow analysis in our compiler stems from this first phase. These include failures to decide the reference kinds (e.g., use/def) for expressions, failures to consider all possible side effects of function calls, failures to correctly update dataflow facts after a program is modified, etc.

With IIR, we can describe dataflow analysis directly on an IR. Since we already have all IR entities uniquely named, we do not need to prepare a naming schema for each analysis. In addition, since all IR entities are already in a predicate form, we can dispense with the second (extraction) phase in many cases. This is a highly desirable feature, particularly in production compilers where each IR entity has many attributes that need to be handled. For instance, when we describe an alias analysis, we need a way to treat some implementation-specific attributes related to memory accesses such as type qualifiers, compiler directives, and options designating physical data layouts. Translating such attributes to an appropriate form for each analysis is tedious and error-prone work, and updating the translation to reflect the modification of IR specifications is often neglected.

As a preliminary case study, we have described a full reaching definition analysis on MIR. Compared to that of Ullman above, our analysis needs no externally given facts. All facts are either IIR entities or generated by analysis. Furthermore, while

Ullman's analysis only treats scalar variables without aliases, our analysis can treat all kinds of expressions including pointer indirections, arrays and their aliases in a similar way.

Here for simplicity we show a statement-wise reaching definition analysis, not a basic block-wise one. The dataflow equation of the analysis is completely standard.

$$
\begin{aligned}
RD_{in}(S) &= \bigcup_{S' \in pred(S)} RD_{out}(S') \\
RD_{out}(S) &= RD_{gen}(S) \cup (RD_{in}(S) - RD_{kill}(S)) \\
RD_{gen}(S) &= \{(S, X) | X \text{ is defined in } S\} \\
RD_{kill}(S) &= \{(?, X) | X \text{ is defined in } S\}
\end{aligned}
$$

For the above equation, we can write our analysis on MIR as follows:

```
rd_in(S,X) :-
  flow(S1,S), rd_out(S1,X).
rd_out(S,X) :-
  rd_gen(S,X) ;
  rd_in(S,X), not(rd_kill(S,X)).
rd_gen(S,X) :-
  def_in_stmt(S,X), scalar(X).
rd_kill(S,X) :-
  def_in_stmt(S,Y), must_alias(X,Y).
```

where `rd_in(S,X)` means that a definition of a variable having an identifier `X` may reach to a statement having an identifier `S`. `rd_out(S,X)`, `rd_gen(S,X)` and `rd_kill(S,X)` have similar meanings. `scalar(X)` means that an expression `X` is a scalar variable. `def_in_stmt(S,X)` means that an expression `X` is in a defining position in a statement `S`, described as follows:

```
def_in_stmt(S,X) :-
  mircode(S,kind,assign),
  ith_child(S, 1, X); ...
```

Finally, `must_alias(X,Y)` means that an expression `X` is definitely aliased with `Y`, which is calculated by an alias analysis whose details we omit here.

These two analyses clearly have the very similar structures, except that the meaning of the reaching definition is more refined in its details in the latter. That is,

- What kinds of expression are regarded as *definitions*? (by `scalar(X)`)
- In what contexts can expressions become *definitions*? (by `def_in_stmt(S,X)`)
- What *definitions* are killed by a defining expression? (by `must_alias(X,Y)`)

All of them can be defined directly based on IIR, where we can retrieve all information available from a specific IR; this is not the case for an analysis based on abstract analysis models.

The total number of lines of the full analysis is under 50. In our implementation, the number of lines of C code for the reaching definition analysis is 2,639. While direct comparison is meaningless, of course, we can say that this declarative description of the analysis is a concise specification of an actual implementation.

We can run the above analysis on an actual MIR generated from C/C++ programs. To ensure termination, we need to use languages with a bottom-up or memoized evaluation strategy such as XSB [33]. Evaluation of the analysis efficiency, however, is beyond the scope of this paper.

4.2 IR Externalization

In a sense, we can view IIR as a means of IR externalization. Although an IR is originally an internal data structure in a compiler, externalization of an IR — that is, outputting an IR outside of a compiler — is often very useful.

IR externalization provides various benefits. The simplest is that it facilitates the debugging of compiler modules. If there is something wrong with a code generated from a compiler, we usually dump IRs into files at several points of a compilation sequence and try to find the module causing the bug by comparing the externalized IRs.

The implementation-independence property of IIR leads to a straightforward and complete IR externalization. The dump format is usually a tree representation of an IR. Such representations are readable but informal, non-uniform and incomplete. In contrast, we can externalize a full IR in an IIR model completely and uniformly.

This completeness and uniformity enable many applications regarding an externalized IR. For example, we can analyze and optimize an externalized IR through auxiliary tools and then put it back into a compiler. We can develop each tool in a different language. This interoperatability makes an IIR a good IR format candidate for a compiler framework such as SUIF.

Another possible application is a differential compiling. By storing a previously compiled IR and a code generated from it in a database, we can save the time needed to recompile the same source program. Developing such a system is not a trivial task in a typical compiler, but with an IIR we can implement this in an obvious way. As an IIR schema instance naturally fits a three-attribute relation of RDB, we can use standard RDBMS to store and load an IR.

5 Discussion

In this section, we discuss design decisions we have made for IIR and compare IIR with a well-known relational data model, which has strong influence on our design.

5.1 IIR Design Decisions

In Section 1, we characterized IIR by two essential conditions:

1. all entities have an explicit identifier.
2. all entities are represented as relations.

Here we discuss these conditions and compare them with other possible definitions.

Condition 1 Condition 1 is the heart of IIR. With explicit identifiers in IIR, we can model virtually all kinds of data structure — such as list, array, tree and graph — used in compiler implementations. One important implication of this condition is that we can now clearly distinguish *entities themselves* with *references to them*. This distinction is obvious at a low-implementation level (memory cells and their addresses), but less clear at an abstract (data model) level.

However, at the cost of this expressiveness, IIR loses some desirable properties which a conventional algebraic data model has. For instance, we cannot use structural induction directly on expressions, which are represented as a set of tuples in an IIR model. That is because the entities of IIR are not constructed inductively.

To compensate for this deficiency, we could make an alternative design decision to only use identifiers when they are indispensable (e.g., to represent control flow graphs). This ad hoc approach is commonly used when treating references in an algebraic data model.

Instead of resorting to such an ad hoc approach, though, we decided to compensate for this problem by specifying well-formedness conditions explicitly. Thanks to condition 2 of IIR, we can use conventional logic programming languages to specify well-formedness conditions in a succinct way. We believe that the merits of expressiveness and uniformity pay for this additional work.

Condition 2 Even after we decided to adopt condition 1, there was still room for alternative design decisions. For example, we could formalize an IR based on functions instead of relations such as IIR. In this approach, an IR is defined as a set of functions of Identifiers to Values, each representing one attribute. This approach provides us with a simpler model than IIR in the following sense:

1. The definition of a function ensures that an attribute a of an identifier i has at most one value $a(i)$. In IIR schema, we need a condition to establish this property.
2. We do not need to inject attributes of different types into one union value. We can use well known rich type systems in studies of functional languages to define the well-typedness of an IR.

Though these properties seem attractive, we decided not to take this approach because of the following drawbacks:

1. The functional model limits the direction of access. In IIR, we can select a set of tuples (i, a, v) based on arbitrary conditions while the only access direction in the functional model is from an identifier and an attribute to a value.
2. As functions are not first-order concepts, we cannot treat an IIR straightforwardly in conventional first-order logic programming languages.

Which is preferable may depend on conditions and applications, but it is fair to say that IIR is a more general data model than the functional one. Similar arguments have been discussed in the database area: e.g., when comparing a network data model and a relational data model.

For our purpose at least, the feasibility of using declarative logic programming languages to write several validity conditions, such as well-typedness, with IIR is a crucial factor. Of course, we can also write a type system in functional languages, but the description may be more operational, unnecessarily

specific, and less readable.

5.2 Comparison with the Relational Data Model

As mentioned, there is a very close connection between IIR and the relational data model (RDM) [9]. Here we discuss and compare these models.

A relation schema R consists of a set of attribute names A, a set of values V and a mapping dom which assigns each attribute a in A to its domain $dom(a) \subseteq V$.

In a rough way, we can view an IIR schema R as a relation schema as follows:

$$A = \{\texttt{identifier}, \texttt{attribute}, \texttt{value}\}$$

$$\begin{array}{lcl} dom(\texttt{identifier}) & = & I^R \\ dom(\texttt{attribute}) & = & A^R \\ dom(\texttt{value}) & = & V^R \end{array}$$

An IIR schema instance can also be viewed as a relation (or a mapping from attributes to values) in an obvious way.

Of course, this simple relational view is too coarse since it cannot exclude ill-formed relations, such as those violating condition 2 of the definition of an IIR instance. This view only enables us to define a proper superset of an IIR instance in RDM.

In the early stages of this work, we used RDM to model an IR. However, the attributes of the relation schema for IR entities were not like the simple ones described above. We followed the conventional means of data modeling in RDM, and modeled each attribute of an IR entity as an attribute of a relation schema for that IR entity. For instance, for the expressions in example 1 in Section 2, we defined three relation schemas — `con`, `var` and `add` — one for each expression class. The attributes of each relation schema are attributes defined in the example, plus one distinguished attribute `identifier`.

$$\begin{array}{lcl} A_{\texttt{con}} & = & \{\texttt{identifier}, \texttt{kind}, \texttt{value}\} \\ A_{\texttt{var}} & = & \{\texttt{identifier}, \texttt{kind}, \texttt{name}\} \\ A_{\texttt{add}} & = & \{\texttt{identifier}, \texttt{kind}, \\ & & \texttt{child1}, \texttt{child2}\} \\ dom(\texttt{identifier}) & = & I^E \\ dom(\texttt{child1}) & = & I^E \\ dom(\texttt{child2}) & = & I^E \\ dom(\texttt{kind}) & = & C^E \\ dom(\texttt{value}) & = & \mathit{Integer} \\ dom(\texttt{name}) & = & \mathit{String} \end{array}$$

In these relation schemas, an expression `(x + x) + 100` can be represented as the following relation instances.

$$\begin{array}{lcl} r_{\texttt{con}} & = & \{(3, \texttt{con}, 100)\} \\ r_{\texttt{var}} & = & \{(4, \texttt{var}, \text{“}x\text{”}), (5, \texttt{var}, \text{“}x\text{”})\} \\ r_{\texttt{add}} & = & \{(1, \texttt{add}, 2, 3), (2, \texttt{add}, 4, 5)\} \end{array}$$

Comparing to IIR, the main differences of this RDB-based approach are as follows:

1. Each expression is modeled as one tuple in a relation.
2. The number of attributes of each relation varies.

The first point allows more compact representation than IIR. In an IIR instance, each entity is represented as a set of tuples, each tuple representing a value of one attribute. For example, entity 2 in $r_{\texttt{var}}$ is represented as three tuples in IIR (see example 1). This entity-as-a-set-of-tuples representation necessitates checking of an extra condition (condition 2 of definition 2) to ensure the functional dependence: i.e., the condition that for each pair of identifiers and attributes there can exist at most one value.

However, the RDM-based approach has drawbacks which eventually led us to abandon it.

- First, as mentioned, IR entities in production compilers usually have many attributes. As a result, in this RDM-based approach, the arity of tuples tends to be larger (over a hundred in our case). Such lengthy tuples are very inconvenient when treating IR instances in conventional logic programming languages like Prolog.
- Second, when we add new attributes to an entity (this is often the case with compilers), we need to change the arity of the relation schema

of that entity. If we refer to tuples of that relation explicitly in the specification, we must change the specification accordingly.

- Third, some IR entities, such as code tree nodes, are classified into many classes (again, over a hundred in our case). To define relation schemas for so many classes is a tedious task.
- Fourth, each value in tuples is anonymous in a relation instance, while in an IIR instance each value is always paired with an attribute name. This makes it much easier to read specific instances, for example when debugging the specifications of test programs. (This point is similar to XML documents.)

To sum up, while RDM is a generic data model which can be applied in a variety of areas, we believe that IIR is more suitable form of modeling in the area of compiler IR, particularly when used with declarative languages.

6 Related Work

In this section, we compare our work with previous works classified into four important aspects of IIR, that is, IR models, identifiers, logic programming languages, and executable specification.

6.1 IR models

Since IIR is an IR data model, direct comparison to each specific IR is impossible. Instead we discuss two IR models (not specific IRs), one based on an object-oriented model and the other on XML.

Object-Oriented IR Modeling IR entities as objects in an object-oriented model is a natural and pervasive concept. Here we take SUIF [22] as a typical example of an object-oriented IR. SUIF is a compiler infrastructure designed to support research in optimizing compilers, and is widely used, particularly in parallel computing.

In SUIF, all IR entities are represented as C++ classes. A whole IR is defined as a class hierarchy starting from a root class called `Object`. For example, all IR statements are defined as subclasses of the `Statement` class. The `Statement` class is a descendant of the `Object` class, inheriting it through the `ExecutionObject` class, the `ScopedObject` class, the `AnnotatableObject` class, and the `SuifObject` class to the `Object` class. As each IR entity is represented as a C++ class, its attributes are defined as members of its class. For instance, expression trees are constructed using (physical) pointer members to designate each tree's children or parent.

Compared to SUIF, an IR based on an IIR model can be defined completely in an implementation-independent way as discussed in Section 5. The syntax and semantics are based on mathematics more succinct and simpler than for C++. IIR can also support inheritance, which is one of the most notable merits of an object-oriented approach, through the usual set inclusion relation on attributes. Furthermore, as entities in SUIF are defined at a class level, not an instance level, the validity condition of IR cannot be specified as precisely as that of IIR.

To help users of SUIF develop their own IR, SUIF provides a high-level specification language called Hoof. The SUIF Macro Generator (smgn) translates Hoof representations to C++ class definition files (.h and .cpp). The Hoof representation hides some implementation details. Constructors, getter/setter methods, and so on are automatically generated by smgn from simpler class definition in Hoof. However, as the name Macro Generator suggests, this is essentially syntax-sugared C++ and inseparably dependent on the syntax and semantics of C++.

XML IR Recent work has been done on defining an IR in XML [15]. This is also a natural idea as major IR entities such as expressions are tree-structured data with attributes, which fit in well with an XML format. The merits of an XML-based approach are as follows:

1. XML IR is completely implementation independent. We can use any language to implement our compiler.
2. We can define a specific IR based on a well-defined, well known XML data model. This lessens the burden of developing the formal definition of an IR from scratch. We can define the well-formedness of our IR by defining DTD (Document Type Definition) or XML schemas.

Compared to IIR, an XML-based approach also has drawbacks. While an XML-based approach overcomes the problem of dependence on a specific implementation languages, XML does not fit in well with graph-structured data like control flow graphs. In addition, the validity conditions are also defined

at a schema level rather than an instance level, and so they are inevitably imprecise.

6.2 Identifiers

The concept of explicit identifiers of entities in IIR is closely related to the issue of object identification in the area of database.

Wieringa and de Jonge discuss this issue comprehensively [40]. They give a precise definition of object identifiers and distinguish it from keys and internal identifiers (designating surrogates).

The major conditions of object identifiers are as follows:

1. non-updatable
2. unique across all possible states of the world
3. visible to the user (i.e. explicitly represented)

Condition 2 is needed since Wieringa and de Jonge discuss a general data modeling issue, in which state changes (i.e., database updates) must be considered. Keys differ from object identifiers in that they are updatable and are unique only in each single state of a database (or a relation). In addition, internal identifiers differ from object identifiers in that they are not visible to the user (i.e. they are implicitly represented).

Identifiers in IIR must satisfy condition 3 by the definition of IIR. Conditions 1 and 2 are not necessary, but if they are satisfied, we can trace each IR entity through all of the transformation processes done by compilers. This property may be beneficial when verifying the correctness of transformation: for example, to resolve the branch path matching problem discussed regarding a certified compiler by Necula [24]. Further discussion of this, however, is beyond the scope of this paper.

Condition 2 is also beneficial when we merge IR entities in different translation units (files). For example, suppose we do a function-inlining optimization over files. If condition 2 is satisfied, we do not need to rename identifiers to avoid name clash. A simple way to satisfy condition 2 is to add a scope-wide unique prefix to identifiers, though this will mean we have to give up an efficient integer representation of identifiers. Evaluation to find the most suitable way for various optimizations remains as future work.

6.3 Using logic languages for compilers

A large amount of effort has gone into the use of logic programming languages to develop code generations, optimizations and analyses in a compiler. Here we only show three works based on Prolog or its relatives, which are used in our work.

The ProCos project directed by Hoare, Jifeng and Bowen tackled the issue of developing provably correct systems [19]. As part of the project, they developed a prototype compiler in Prolog which is quite close to its formal specification. The source language is a simple structured imperative language and the target is a RISC-like assembly language. Statements and expressions are represented as conventional tree-structured terms. Dawson et al. showed a groundness analysis of logic programs and a strictness analysis of functional programs using a Prolog-like language, XSB [11]. Pop et al. have developed a method to write some loop optimizations of GCC4 in a declarative way using Prolog [26]. Their IR is a three-address SSA representation called GIMPLE.

While these methods vary, as for IR, all are based on IR where each entity is simplified or abstracted and ordinarily anonymous. Complex data structures like trees or lists are represented as similar-structured terms in their languages. For example, in [26] each GIMPLE statement is represented as a Prolog term: e.g., an assignment statement `x = y + z` is represented as `assign(x, y + z)`.

As discussed in Section 5, applying methods based on such IR models to a large-scale IR of a production compiler is not a trivial task. It is hard to describe the large number of attributes for each entity (e.g., a type of `x`). Annotating an entity with its attributes may change the arity of the entity, which requires that existing codes be modified accordingly. An interesting research direction, though, would be to develop these methods based on IIR and evaluate the outcome regarding aspects such as efficiency.

6.4 Executable specification

Since writing a formal specification is an error-prone job, specification executability is highly desirable. We can test an executable specification on actual data and confirm that it actually captures the intended meaning by executing a specification and comparing the result with that of an implemen-

tation. This also allows incremental development of a specification, where we begin with a coarse specification and obtain an exact one through a refinement process of checking and modifying. In Section 3, we took this approach to develop a specification of our compiler IR from an existing implementation.

Much work has been done on executable specification. Here we discuss two efforts that are closely related to our work.

Gordon et al. developed a formal semantics of a typed IL (called Baby IL) which is a subset of Microsoft .NET CIL [16]. They use a conventional tree-structured, but slightly tricky, applicative syntax to represent their IL and formalize static and dynamic semantics of it based on HOL [17]. They also provide mechanized proofs of some properties of their IL such as type soundness. While their results are stronger than ours, their work is subject to a limitation in that there is a substantial gap between Baby IL (their model) and CIL (the actual IL), thus their results are inevitably incomplete. As they state, extending their model to full CIL is not a trivial task. An interesting open question is whether the IIR model can straightforwardly represent full CIL and also allow their proof strategy to prove similar results mechanically.

Bishop et al. have developed an executable specification of the TCP/IP protocol and socket API in HOL [5]. They experimentally confirmed the correctness of their specification by sampling behavior traces from several de facto standard implementations and checking that these were contained in the set of traces allowed by the specification. Note that their intention was not to verify implementations against the specification, but to verify the specification against implementations. In other words, they extracted the specification from implementations through a refinement process as in our work described in Section 3. Their work is not aimed at IR, but we share their conclusion regarding the effectiveness of an executable specification in reverse-engineering a specification from an existing implementation.

7 Conclusion

In this paper, we have presented a new IR data model called IIR. The essence of IIR is that

1. All entities have explicit identifiers.
2. All entities are represented as relations.

Condition 1 ensures that every IR entity can be referred to by its identifier. By using identifiers as symbolic pointers we can model actual structures of IR faithfully in IIR. Condition 2 makes it possible to treat IR within declarative frameworks such as logic programming languages. Based on these, we could develop a sufficiently precise specification of the IR of our production compiler.

As shown in Section 4, IIR enables broader applications. One example is IR externalization. By using IIR as an implementation-independent IR format, we can use *any* implementation language to develop each module of a compiler. While XML has been used in some studies to partially realize this interoperatability, IIR is more general and expressive, and fits well with logic programming languages, which we believe are more suitable for treating IR compared to, for example, XSLT.

As another example, we have shown that dataflow analysis of reaching definitions can be easily written with IIR. Many ways of developing program analyses, optimizations and code generations in declarative frameworks have been proposed, and these techniques are also applicable to IIR in a straightforward way.

Future Works Many areas remain to be explored. One direction is to seek more effective ways of developing a specification by exploiting several properties of IIR. For example, while attribute names are first-class values in IIR, we did not fully exploit this property in our work. This property allows us to quantify over attribute names in first-order logic, which may provide several benefits like the row-types in ML [27].

Another direction is to develop a variety of common formal systems of IR on IIR. In this paper, we concentrated on the static semantics (syntax and types) of IR and showed that we could easily define the well-typedness of the IR of our production compiler based on IIR. We have not yet defined the dynamic semantics, but we anticipate no serious difficulty in this. By using identifiers of code tree nodes as control points, we can straightforwardly define the operational semantics of code trees. The ASM-based approach taken by Stark et al. [32], where they use path indices to identify each syntactic element in expressions, is probably also appli-

cable in this regard. Denotational semantics based approach for IR used in e.g., [20] [1] should also be applicable.

The property of IIR that all expressions (not restricted to variables) have unique identifiers reminds us of some well-known style/form such as CPS (Continuation Passing Style) [3], ANF (A Normal Form) [14], or SSA (Static Single Assignment) form [10]. Although we have not yet appreciated how to relate IIR with these forms, it may be possible to exploit identifiers of IIR when converting a source program to these forms.

We also plan to implement several dataflow analyses based on IIR. For example, we believe our previous work of pointer analysis [7] [8] can be easily reformalized based on IIR. This will make it possible to prove the correctness of the analysis implementation itself, rather than the abstracted specification.

Throughout this work, we have sought to find a way of specifying all concerns about IR in a declarative way, which we believe is the best approach to taming the chaotic situations that arise in actual software developments. We hope that our work will help bridge the gap between research and practice, and reduce the burden of developing modern production compilers, which are complex, large, and thus error-prone.

Acknowledgement We gratefully acknowledge helpful feedback from the PPL referees.

References

[1] Abe, S., Hagiya, M. and Nakata, I.: A Retargetable Code Generator for the Generic Intermediate Language in COINS, *IPSJ Transactions on Programming*, Vol. 46, No. SIG14(2005).

[2] Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: principles, techniques, and tools*, Addison-Wesley, 1986.

[3] Appel, A. W.: *Compiling with continuations*, Cambridge University Press, New York, NY, USA, 1992.

[4] Assmann, U.: How to uniformly specify program analysis and transformation with graph rewrite systems, in *Proceedings of the 6th International Conference on Compiler Constructions*, 1996, pp. 121–135.

[5] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M. and Wansbrough, K.: Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations, in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp. 55–66.

[6] Carr, S. and Kennedy, K.: Scalar replacement in the presence of conditional control flow, Technical Report TR92283, Rice University, CRPC, 1992.

[7] Chishiro, E.: Pointer Analysis for Type-unsafe C Programs, *IPSJ Transactions on Programming*, Vol. 45, No. SIG12(2004), pp. 52–66.

[8] Chishiro, E.: Pointer Analysis for C Programs Using a Deductive System, *IPSJ Transactions on Programming*, Vol. 47, No. SIG2(2006), pp. 1–17.

[9] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks, *Communications of the ACM*, Vol. 13, No. 6(1970), pp. 377–387.

[10] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. and Zadeck, F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4(1991), pp. 451–490.

[11] Dawson, S., Ramakrishnam, C. R. and Warren, D. S.: Practical Program Analysis Using General Purpose Logic Programming Systems — a Case Study, in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996, pp. 117–126.

[12] Drape, S., de Moor, O. and Sittampalam, G.: Transforming the .NET Intermediate Language Using Path Logic Programming, in *Proceedings of the ACM Conference on Principles and Practice of Declarative Programming*, 2002, pp. 133–144.

[13] Embedded Microprocessor Benchmark Consortium: http://www. eembc.org, 2000.

[14] Flanagan, C., Sabry, A., Duba, B. F. and Felleisen, M.: The Essence of Compiling with Continuations, in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993, pp. 237–247.

[15] Germon, R.: Using XML as an Intermediate Form for Compiler Development, in *XML Conference and Exposition*, 2001.

[16] Gordon, A. D. and Syme, D.: Typing a multi-language intermediate code, in *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001, pp. 248–260.

[17] Gordon, M. J. C. and T. Melham, E.: *Introduction to HOL: a theorem proving environment*, Cambridge University Press, 1993.

[18] Horwitz, S., Reps, T. and Sagiv, M.: Demand Interprocedural Dataflow Analysis, in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995, pp. 104–115.

[19] Jifeng, H.: *Provably Correct Systems: Modelling of Communication Languages and Design of Optimized Compilers*, McGraw-Hill International, 1995.

[20] Jones, S. L. P.: Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine, *Journal of Functional Programming*, Vol. 2, No. 2(1992), pp. 127–202.

[21] Jones, S. L. P., Hall, C., Hammond, K., Partain, W. and Wadler, P.: The Glasgow Haskell compiler: a technical overview, in *Proceedings of Joint Framework for Information Technology Technical Conference*, 1993, pp. 249–257.

[22] Lam, M. S.: An Overview of the SUIF2 System, in *A Tutorial of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[23] Morrisett, G., Walker, D., Crary, K. and Glew, N.: From system F to typed assembly language, *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 3(1999), pp. 527–568.

[24] Necula, G. C.: Translation validation for an optimizing compiler, in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 83–94.

[25] Papaspyrou, N. S.: A Formal Semantics for the C Programming Language, PhD thesis, National Technical University of Athens, Software Engineering Laboratory, 1998.

[26] Pop, S., Cohen, A., Jouvelot, P. and Silber, G.-A.: The New Framework for Loop Nest Optimization in GCC: from Prototyping to Evaluation, in *Proceedings of the Compilers for Parallel Computers*, 2006.

[27] Remy, D.: Typechecking records and variants in a natural extension of ML, in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1989, pp. 77–88.

[28] Renesas Technology Corporation: The C/C++ Compiler Packages for the SuperH RISC engine family Revised to Their V.9.00, http://documentation.renesas.com/eng/products/tool/tu/tncsxa089ae.pdf, 2005.

[29] Renesas Technology Corporation: RENESAS TOOL NEWS 051101D, http://tool-support.renesas.com/eng/toolnews/n051101/tn6.htm, 2005.

[30] Reynolds, J. C.: The Essence of Algol, *Algorithmic Languages*, de Bakker, J. W. and van Vliet, J. C.(eds.), 1981, pp. 345–372.

[31] Standard Performance Evaluation Corporation: http://www.spec.org, 1995.

[32] Stark, R. F., Borger, E. and Shmid, J.: *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag, 2001.

[33] Stony Brook University: XSB, http://xsb.sourceforge. net, 1990.

[34] SWI-Prolog Foundation: SWI-Prolog, http://www.swi-prolog.org, 2004.

[35] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R. and Lee, P.: TIL: a type-directed optimizing compiler for ML, in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996, pp. 181–192.

[36] Ullman, J. D.: *Principles of Database and Knowledge-base Systems*, Computer Science Press, 1989.

[37] van Leeuwen, J.(ed.): *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, Elsevier and MIT Press, 1990.

[38] Whaley, J. and Lam, M. S.: Cloning-based Context-sensitive Pointer Analysis Using Binary Decision Diagrams, in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004, pp. 131–144.

[39] Whitfield, D. L. and Soffa, M. L.: An approach for exploring code improving transformations, *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 6(1997), pp. 1053–1084.

[40] Wieringa, R. and de Jonge, W.: Object Identifiers, Keys, and Surrogates: Object Identifiers Revisited, *Theory and Practice of Object Systems*, Vol. 1, No. 2(1995), pp. 101–114.

[41] Winskel, G.: *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, 1993.

A Rules In Prolog

In this section, we explain informally the syntax and semantics of rules used in this paper. See e.g., [37] for detail about logic programming.

The general form of rules is as follows:

```
p :- q11, q12, ...
  ;  q21, q22, ...
  ...
  ;  qn1, qn2, ....
```

where `p` and `q`ij are atomic formulas. You can read this as a formula in first order logic by translating

- `,` to $\wedge$,
- `;` to $\vee$,
- `:-` to $\Leftarrow$

and universally quantifying all variables in `p` and `q`ij. In atomic formulas, all variables begin with a capital letter and can be distinguished from constants.

For example,

```
p(X) :- q(X,Y)
     ;  r(X).
```

is interpreted as

$$\forall X, Y.p(X) \Leftarrow (q(X,Y) \vee r(X))$$

or equivalently,

$$\forall X.((\exists Y.q(X,Y)) \vee r(X)) \Rightarrow p(X)$$

When validity conditions of IR are described as a set of rules, these rules are interpreted as a conjunction of formulas for each rules.

B The Relational Data Model

In this section, we briefly explain the definitions of relational data model used in this paper. See e.g., [37] for detail.

We call the set of all attributes of entities the *universe* U. An attribute A is a name, related with a set of values (called a *domain* of A). A *relation schema* R is a tuple (an ordered subset) †3 of U. We call $|R|$ as an arity of R. A *database schema* D over U is a set of relation schemas.

For each schema, we can define its *instance*. Let $R_i = (A_{i1}, ..., A_{i|R_i|})$ be a relation schema, and $D = \{R_1, ..., R_m\}$ be a database schema. A *relation instance* r_i (or short, relation) over R_i is a set of tuples $(v_1, ..., v_{|R_i|})$ where each v_j is in the domain of A_{ij}. A *database instance* (or short, database) over D is a set of relations $\{r_1, ..., r_m\}$ where each r_i is a relation over R_i.

Based on these definitions, we can define well-known primitive operations over relations such as projection, join, union, difference and selection, though we omit details here.

For example, we can model a directed graph of integers as following schemas:

$$\begin{aligned} D_{\mathtt{graph}} &= \{R_{\mathtt{node}}, R_{\mathtt{edge}}\} \\ R_{\mathtt{node}} &= (\mathtt{number}) \\ R_{\mathtt{edge}} &= (\mathtt{from}, \mathtt{to}) \end{aligned}$$

where domains of `number`, `from` and `to` are integers. The below is a possible instance of $D_{\mathtt{graph}}$:

$$\begin{aligned} d_{\mathtt{graph}} &= \{r_{\mathtt{node}}, r_{\mathtt{edge}}\} \\ r_{\mathtt{node}} &= \{(1), (2), (3), (4)\} \\ r_{\mathtt{edge}} &= \{(1,2), (1,3), (2,4), (3,4)\} \end{aligned}$$

†3 It is more general to define a relation schema as a subset of U, and a relation instance as a function from attributes to values accordingly, though the above definition is suffice for this paper.